Expression Language Specification

Version 3.0 Final Release

Kin-man Chung, editor

Oracle Corporation
www.oracle.com

Final Release - April 29, 2013

Send comments to: users@el-spec.java.net

Oracle Corporation
www.oracle.com

Send comments to: users@el-spec.java.net

Specification: JSR-341 Expression Language

Version: 3.0

Status: Final Release

Release: April 29, 2013

Contents

Preface

This is the Expression Language specification version 3.0, developed by the JSR-341 (EL 3.0) expert groups under the Java Community Process. See http://www.jcp.org.

Historical Note

The EL was originally inspired by both ECMAScript and the XPath expression languages. During its inception, the experts involved were very reluctant to design yet another expression language and tried to use each of these languages, but they fell short in different areas.

The JSP Standard Tag Library (JSTL) version 1.0 (based on JSP 1.2) was therefore first to introduce an Expression Language (EL) to make it easy for page authors to access and manipulate application data without having to master the complexity associated with programming languages such as Java and JavaScript.

Given its success, the EL was subsequently moved into the JSP specification (JSP 2.0/JSTL 1.1), making it generally available within JSP pages (not just for attributes of JSTL tag libraries).

JavaServer Faces 1.0 defined a standard framework for building User Interface components, and was built on top of JSP 1.2 technology. Because JSP 1.2 technology did not have an integrated expression language and because the JSP 2.0 EL did not meet all of the needs of Faces, an EL variant was developed for Faces 1.0. The Faces expert group (EG) attempted to make the language as compatible with JSP 2.0 as possible but some differences were necessary.

It was obviously desirable to have a single, unified expression language that meets the needs of the various web-tier technologies. The Faces and JSP EGs therefore worked together on the specification of a unified expression language, defined in JSR 245, and which took effect for the JSP 2.1 and Faces 1.2 releases.

The JSP/JSTL/Faces expert groups also acknowledged that the Expression Language(EL) is useful beyond their own specifications. This specification is the first JSR that defines the Expression Language as an independent specification, with no dependencies on other technologies.

Typographical Conventions

Font Style	Uses
Italic	Emphasis, definition of term.
`Monospace`	Syntax, code examples, attribute names, Java language types, API, enumerated attribute values.

Comments

We are interested in improving this specification and welcome your comments and suggestions. We have a java.net project with an issue tracker and a mailing list for comments and discussions about this specification.

Project:

http://java.net/projects/el-spec

Mail alias for comments:

users@el-spec.java.net

CHAPTER 1

Language Syntax and Semantics

The syntax and semantics of the Expression Language (EL) are described in this chapter.

1.1 Overview

The EL was originally designed as a simple language to meet the needs of the presentation layer in web applications. It features:

- A simple syntax restricted to the evaluation of expressions
- Variables and nested properties
- Relational, logical, arithmetic, conditional, and empty operators
- Functions implemented as static methods on Java classes
- Lenient semantics where appropriate default values and type conversions are provided to minimize exposing errors to end users

as well as

- A pluggable API for resolving variable references into Java objects and for resolving the properties applied to these Java objects
- An API for deferred evaluation of expressions that refer to either values or methods on an object
- Support for lvalue expressions (expressions a value can be assigned to)

These last three features are key additions to the JSP 2.0 EL resulting from the EL alignment work done in the JSP 2.1 and Faces 1.2 specifications.

EL 3.0 adds features to enable EL to be used as a stand-alone tool. It introduces APIs for direct evaluation of EL expressions and manipulation of EL environments. It also adds some powerful features to the language, such as the support of operations for collection objects.

1.1.1 EL in a nutshell

The syntax is quite simple. Model objects are accessed by name. A generalized `[]` operator can be used to access maps, lists, arrays of objects and properties of a JavaBeans object, and to invoke methods in a JavaBeans object; the operator can be nested arbitrarily. The `.` operator can be used as a convenient shorthand for property access when the property name follows the conventions of Java identifiers, but the `[]` operator allows for more generalized access. Similarly, . operator can also be used to invoke methods, when the method name is known, but the [] operator can be used to invoke methods dynamically.

Relational comparisons are allowed using the standard Java relational operators. Comparisons may be made against other values, or against boolean (for equality comparisons only), string, integer, or floating point literals. Arithmetic operators can be used to compute integer and floating point values. Logical operators are available.

The EL features a flexible architecture where the resolution of model objects (and their associated properties and methods), functions, and variables are all performed through a pluggable API, making the EL easily adaptable to various environments.

1.2 EL Expressions

An EL expression is specified either as an *eval-expression*, or as a *literal-expression*. The EL also supports *composite expressions*, where multiple EL expressions (eval-expressions and literal-expressions) are grouped together.

An EL expression is parsed as either a *value expression* or a *method expression*. A value expression refers to a value, whereas a method expression refers to a method on an object. Once parsed, the expression can optionally be evaluated one or more times.

Each type of expression (eval-expression, literal-expression, and composite expression) is described in its own section below.

1.2.1 Eval-expression

An eval-expression is formed by using the constructs `${expr}` or `#{expr}`. Both constructs are parsed and evaluated in exactly the same way by the EL, even though they might carry different meanings in the technology that is using the EL.

For instance, by convention the JavaEE web tier specifications use the `${expr}` construct for immediate evaluation and the `#{expr}` construct for deferred evaluation. This difference in delimiters points out the semantic differences between the two expression types in the JavaEE web tier. Expressions delimited by “#{}” are said to use “deferred evaluation” because the expression is not evaluated until its value is needed by the system. Expressions delimited by “${}" are said to use “immediate evaluation” because the expression is compiled when the JSP page is compiled and it is executed when the JSP page is executed. More on this in Section 1.2.4, “Syntax restrictions”.

Other technologies may choose to use the same convention. It is up to each technology to enforce its own restrictions on where each construct can be used.

In some EL APIs, especially those introduced in EL 3.0 to support stand-alone use, the EL expressions are specified without `${}` or `#{}` delimiters.

Nested eval-expressions, such as `${item[${i}]}`, are illegal.

1.2.1.1 Eval-expressions as value expressions

When parsed as a value expression, an eval-expression can be evaluated as either an *rvalue* or an *lvalue*. An *rvalue* is an expression that would typically appear on the right side of the assignment operator. An *lvalue* would typically appear on the left side.

For instance, all EL expressions in JSP 2.0 are evaluated by the JSP engine immediately when the page response is rendered. They all yield rvalues.

In the following JSTL action

```
<c:out value="${customer.name}"/>
```

the expression `${customer.name}` is evaluated by the JSP engine and the returned value is fed to the tag handler and converted to the type associated with the attribute (`String` in this case).

Faces, on the other hand, supports a full UI component model that requires expressions to represent more than just rvalues. It needs expressions to represent references to data structures whose value could be assigned, as well as to represent methods that could be invoked.

For example, in the following Faces code sample:

```
<h:form>
    <h:inputText
      id="email"
      value="#{checkOutFormBean.email}"
      size="25" maxlength="125"
      validator="#{checkOutFormBean.validateEmail}"/>
</h:form>
```

when the form is submitted, the "apply request values" phase of Faces evaluates the EL expression `#{checkOutFormBean.email}` as a reference to a data structure whose value is set with the input parameter it is associated with in the form. The result of the expression therefore represents a reference to a data structure, or an lvalue, the left hand side of an assignment operation.

When that same expression is evaluated during the rendering phase, it yields the specific value associated with the object (rvalue), just as would be the case with JSP.

The valid syntax for an lvalue is a subset of the valid syntax for an rvalue. In particular, an lvalue can only consist of either a single variable (e.g. `${name}`) or a property resolution on some object, via the `.` or `[]` operator (e.g. `${employee.name}`). Of course, an EL function or method that returns either an object or a name can be part of an lvalue.

When parsing a value expression, an expected type is provided. In the case of an rvalue, the expected type is what the result of the expression evaluation is coerced to. In the case of lvalues, the expected type is ignored and the provided value is coerced to the actual type of the property the expression points to, before that property is set. The EL type conversion rules are defined in Section 1.23, "Type Conversion". A few sample eval-expressions are shown in FIGURE 1-1.

Expression	Expected Type	Result
`${customer.name}`	`String`	`Guy Lafleur` Expression evaluates to a String. No conversion necessary.
`${book}`	`String`	`Wonders of the World` Expression evaluates to a `Book` object (e.g. `com.example.Book`). Conversion rules result in the evaluation of `book.toString()`, which could for example yield the book title.

FIGURE 1-1 Sample eval-expressions

1.2.1.2 Eval-expressions as method expressions

In some cases, it is desirable for an EL expression to refer to a method instead of a model object.

For instance, in JSF, a component tag also has a set of attributes for referencing methods that can perform certain functions for the component associated with the tag. To support these types of expressions, the EL defines method expressions (EL class `MethodExpression`).

In the above example, the validator attribute uses an expression that is associated with type `MethodExpression`. Just as with `ValueExpressions`, the evaluation of the expression (calling the method) is deferred and can be processed by the underlying technology at the appropriate moment within its life cycle.

A method expression shares the same syntax as an lvalue. That is, it can only consist of either a single variable (e.g. `${name}`) or a property resolution on some object, via the `.` or `[]` operator (e.g. `${employee.name}`). Information about the expected return type and parameter types is provided at the time the method is parsed.

A method expression is evaluated by invoking its referenced method or by retrieving information about the referenced method. Upon evaluation, if the expected signature is provided at parse time, the EL API verifies that the method conforms to the expected signature, and there is therefore no coercion performed. If the expected signature is not provided at parse time, then at evaluation, the method is identified with the information of the parameters in the expression, and the parameters are coerced to the respective formal types.

1.2.2 Literal-expression

A literal-expression does not use the `${expr}` or `#{expr}` constructs, and simply evaluates to the text of the expression, of type `String`. Upon evaluation, an expected type of something other than `String` can be provided. Sample literal-expressions are shown in FIGURE 1-2.

Expression	Expected Type	Result
`Aloha!`	`String`	`Aloha!`
`true`	`Boolean`	`Boolean.TRUE`

FIGURE 1-2 Sample literal-expressions

To generate literal values that include the character sequence “`${`" or “`#{`“, the developer can choose to use a composite expression as shown here:

${**'${'}**exprA}

#{'#{'**}**exprB}The resulting values would then be the strings `${exprA}` and `#{exprB}`.

Alternatively, the escape characters `\$` and `\#` can be used to escape what would otherwise be treated as an eval-expression. Given the literal-expressions:

```
\${exprA}
\#{exprB}
```

The resulting values would again be the strings `${exprA}` and `#{exprB}`.

A literal-expression can be used anywhere a value expression can be used. A literal-expression can also be used as a method expression that returns a non-void return value. The standard EL coercion rules (see Section 1.23, "Type Conversion") then apply if the return type of the method expression is not java.lang.String.

1.2.3 Composite expressions

The EL also supports *composite expressions*, where multiple EL expressions are grouped together. With composite expressions, eval-expressions are evaluated from left to right, coerced to `String`s (according to the EL type conversion rules), and concatenated with any intervening literal-expressions.

For example, the composite expression "`${firstName} ${lastName}`" is composed of three EL expressions: eval-expression "`${firstName}`", literal-expression " ", and eval-expression "`${lastName}`".

Once evaluated, the resulting `String` is then coerced to the expected type, according to the EL type conversion rules. A sample composite expression is shown in FIGURE 1-3.

Expression	Expected Type	Result
`Welcome ${customer.name} to our site`	`String`	`Welcome Guy Lafleur to our site` `${customer.name}` evaluates to a String which is then concatenated with the literal-expressions. No conversion necessary.

FIGURE 1-3 Sample composite expression

It is illegal to mix `${}` and `#{}` constructs in a composite expression. This restriction is imposed to avoid ambiguities should a user think that using `${expr}` or `#{expr}` dictates how an expression is evaluated. For instance, as was mentioned previously, the convention in the J2EE web tier specifications is for `${}` to mean immediate evaluation and for `#{}` to mean deferred evaluation. This means that in EL expressions in the J2EE web tier, a developer cannot force immediate evaluation of some parts of a composite expression and deferred evaluation of other parts. This restriction may be lifted in future versions to allow for more advanced EL usage patterns.

For APIs prior to EL 3.0, a composite expression can be used anywhere an EL expression can be used except for when parsing a method expression. Only a single eval-expression can be used to parse a method expression.

Some APIs in EL 3.0 use only single eval-expressions, and not the composite expressions. However, there is no lost in functionality, since a composite expression can be specified with a single eval-expressions, by using the string concatenation operators, introduced in EL 3.0. For instance, the composite expression

```
Welcome ${customer.name} to our site
```

can be written as

```
${‘Welcome ‘ += customer.name += ‘ to our site’}.
```

1.2.4 Syntax restrictions

While `${}` and `#{}` eval-expressions are parsed and evaluated in exactly the same way by the EL, the underlying technology is free to impose restrictions on which syntax can be used according to where the expression appears.

For instance, in JSP 2.1, `#{}` expressions are only allowed for tag attributes that accept deferred expressions. `#{expr}` will generate an error if used anywhere else.

1.3 Literals

There are literals for boolean, integer, floating point, string, and null in an eval-expression.

- Boolean - `true` and `false`
- Integer - As defined by the `IntegerLiteral` construct in Section 1.24
- Floating point - As defined by the `FloatingPointLiteral` construct in Section 1.24

- String - With single and double quotes - `"` is escaped as `\"`, `'` is escaped as `\'`, and \ is escaped as `\\`. Quotes only need to be escaped in a string value enclosed in the same type of quote
- Null - `null`

1.4 Errors, Warnings, Default Values

The Expression Language has been designed with the presentation layer of web applications in mind. In that usage, experience suggests that it is most important to be able to provide as good a presentation as possible, even when there are simple errors in the page. To meet this requirement, the EL does not provide warnings, just default values and errors. Default values are type-correct values that are assigned to a subexpression when there is some problem. An error is an exception thrown (to be handled by the environment where the EL is used).

1.5 Resolution of Model Objects and their Properties or Methods

A core concept in the EL is the evaluation of a model object name into an object, and the resolution of properties or methods applied to objects in an expression (operators `.` and `[]`).

The EL API provides a generalized mechanism, an `ELResolver`, implemented by the underlying technology and which defines the rules that govern the resolution of model object names and their associated properties.

The resolution of names and properties is further affected by the presence of

- Functions. See Section 1.18, “Functions”.
- Variables. See Section 1.19, “Variables”.
- Imported names (classes, fields, and methods). See Section 1.22, “Static Field and Method Reference”.
- Lambda expressions and arguments. See Section 1.20, “Lambda Expressions”.

The rules described below are used in resolving names and properties when evaluating identifiers, function calls, and object properties and method calls.

1.5.1 Evaluating Identifiers

The steps are used for evaluating an identifier.

- If the identifier is a lambda argument passed to a lambda expression invocation, its value is returned.
- Else if the identifier is a variable, the associated expression is evaluated and returned.
- Else if the identifier is resolved by the `ELResolvers`, the value returned from the `ELResolvers` is returned.
- Else if the identifier is an imported static field, its value is returned.
- Else return not resolved.

One implication of the explicit search order of the identifiers is that an identifier hides other identifiers (of the same name) that come after it in the list.

1.5.2 Evaluating functions

The expression with the syntax *func(args...)(args...)...* can mean any of the following.

- A call to an EL fucntion with empty namespace.
- A call to a lambda expression.
- A call to the constructor of an imported class.
- A call to a static method that has been imported statically.

Note the above syntax allows the invocation of a lambda expression that returns another lambda expression, which is then invoked.

The following steps are used to evaluate the above expression.

- Evaluate the name of the function as an identifier.
  - If the identifier is a lambda argument passed to a lambda expression invocation, its value is returned.
  - Else if the identifier is a variable, the associated expression is evaluated and returned.
  - Else if the identifier is resolved by the `ELResolvers`, the value returned from the `ELResolvers` is returned.
- If the result of evaluating the function name is a `LambdaExpression`, the `LambdaExpression` is invoked with the supplied arguments. If the result of evaluating the `LambdaExpression` is another `LambdaExpression`, and the syntax contains repeated function invocations, such as *func()()...*, then the resultant `LambdaExpression` is in turn evaluated, and so on.

- Else if the function has been mapped previously in a `FunctionMapper`, the mapped method is invoked with the supplied arguments.
- Else if the function name is the name of an imported class, the constructor for this class is invoked with the supplied arguments.
- Else if the function name is the name of an imported static method, the method is invoked with the supplied arguments.
- Else error.

1.5.3 Evaluating objects with properties

The steps for evaluating an expression with [] or . operators (property reference and method call) are described in Section 1.6, "Operators [] and .". However, the syntax for . operator is also used to reference a static field, or to invoke a static method. Therefore if the expression with a . operator is not resolved by the `ELResolvers`, and if the identifier for the base object is the name of an imported class, the expression becomes a reference to a static field, or an invocation of a static method, of the imported class.

1.5.4 Invoking method expressions

A method expression can consist of either a single variable (e.g. `${name}`) or a property resolution on some object, via the `.` or `[]` operator (e.g. `${employee.getName}`). Section 1.6, "Operators [] and ." describes how to invoke a method of an object. This form of method expressions allows arguments to the method to be specified in the EL expression (e.g. `${employee.getName()}`).

To invoke a method expression of a single variable, the identifier is first evaluated, as decribed in Section 1.5.1, "Evaluating Identifiers". If the identifier evaluates to a `javax.el.MethodExpression`, the method expression is invoked and the result returned, otherwise an error is raised. This form of method expression does not allow arguments to be specified in the EL expression.

1.6 Operators `[]` and `.`

The EL follows ECMAScript in unifying the treatment of the `.` and `[]` operators.

`expr-a.identifier-b` is equivalent to `expr-a["identifier-b"]`; that is, the identifier `identifier-b` is used to construct a literal whose value is the identifier, and then the `[]` operator is used with that value.

Similarly, `expr-a.identifier-b(params)`is equivalent to `expr-a["identifier-b"](params)`.

The expression `expr-a["identifier-b"](params)`denotes a parametered method invocation, where `params` is a comma-separated list of expressions denoting the parameters for the method call.

To evaluate `expr-a[expr-b] or expr-a[expr-b](params)`:

- Evaluate `expr-a` into `value-a`.
- If `value-a` is `null`:
  - If `expr-a[expr-b]` is the last property being resolved:
    - If the expression is a value expression and `ValueExpression.getValue(context)` was called to initiate this expression evaluation, return `null`.
    - Otherwise, throw `PropertyNotFoundException`. *[trying to de-reference null for an lvalue]*
  - Otherwise, return `null`.
- Evaluate `expr-b` into `value-b` .
- If `value-b` is `null`:
  - If `expr-a[expr-b]` is the last property being resolved:
    - If the expression is a value expression and `ValueExpression.getValue(context)` was called to initiate this expression evaluation, return `null`.
    - Otherwise, throw `PropertyNotFoundException`. *[trying to de-reference null for an lvalue]*
  - Otherwise, return `null`.
- If the expression is a value expression:
  - If `expr-a[expr-b]` is the last property being resolved:
    - If `ValueExpression.getValue(context)` was called to initiate this expression evaluation.
      - If the expression is a parametered method call, evaluate `params` into `param-values`, and invoke `elResolver.invoke(context, value-a, value-b, null, param-values)`.
      - Otherwise, `invoke elResolver.getValue(value-a, value-b)`.
    - If `ValueExpression.getType(context)` was called, invoke `elResolver.getType(context, value-a, value-b)`.
    - If `ValueExpression.isReadOnly(context)` was called, invoke `elResolver.isReadOnly(context, value-a, value-b)`.
    - If `ValueExpression.setValue(context, val)` was called, invoke `elResolver.setValue(context, value-a, value-b, val)`.

    - Otherwise:
        - If the expression is a parametered method call, evaluate `params` into `param-values`, and invoke `elResolver.invoke(context, value-a, value-b, null, params)`.
        - `Otherwise, invoke elResolver.getValue(value-a, value-b).`
- Otherwise, the expression is a method expression:
    - If `expr-a[expr-b]` is the last property being resolved:
        - Coerce `value-b` to `String`.
        - If the expression is not a parametered method call, find the method on object `value-a` with name `value-b` and with the set of expected parameter types provided at parse time. If the method does not exist, or the return type does not match the expected return type provided at parse time, throw `MethodNotFoundException`.
        - If `MethodExpression.invoke(context, params)` was called:
            - If the expression is a parametered method call, evaluate `params` into `param-values`, and invoke `elResolver.invoke(context, value-a, value-b, paramTypes, param-values), where paramTypes is the parameter types, if provided at parse time, and is null otherwise.`
            - Otherwise, invoke the found method with the parameters passed to the invoke method.
        - If `MethodExpression.getMethodInfo(context)` was called, construct and return a new `MethodInfo` object.
    - Otherwise:
        - If the expression is a parametered method call, evaluate `params` into `param-values`, and invoke `elResolver.invoke(context, value-a, value-b, null, params)`.
        - Otherwise, invoke `elResolver.getValue(value-a, value-b)`.

1.7 Arithmetic Operators

Arithmetic is provided to act on integer (`BigInteger` and `Long`) and floating point (`BigDecimal` and `Double`) values. There are 5 operators:

- Addition: `+`
- Substraction: `-`
- Multiplication: `*`

- Division: `/` and `div`
- Remainder (modulo): `%` and `mod`

The last two operators are available in both syntaxes to be consistent with XPath and ECMAScript.

The evaluation of arithmetic operators is described in the following sections. `A` and `B` are the evaluation of subexpressions

1.7.1 Binary operators - `A {+,-,*} B`

- If `A` and `B` are null, return `(Long)0`
- If `A` or `B` is a `BigDecimal`, coerce both to `BigDecimal` and then:
  - If operator is `+`, return `A.add(B)`
  - If operator is `-`, return `A.subtract(B)`
  - If operator is `*`, return `A.multiply(B)`
- If `A` or `B` is a `Float`, `Double`, or `String` containing `.`, `e`, or `E`:
  - If `A` or `B` is `BigInteger`, coerce both `A` and `B` to `BigDecimal` and apply operator.
  - Otherwise, coerce both `A` and `B` to `Double` and apply operator
- If `A` or `B` is `BigInteger`, coerce both to `BigInteger` and then:
  - If operator is `+`, return `A.add(B)`
  - If operator is `-`, return `A.subtract(B)`
  - If operator is `*`, return `A.multiply(B)`
- Otherwise coerce both `A` and `B` to `Long` and apply operator
- If operator results in exception, error

1.7.2 Binary operator - `A {/,div} B`

- If `A` and `B` are null, return `(Long)0`
- If `A` or `B` is a `BigDecimal` or a `BigInteger`, coerce both to `BigDecimal` and return `A.divide(B, BigDecimal.ROUND_HALF_UP)`
- Otherwise, coerce both `A` and `B` to `Double` and apply operator
- If operator results in exception, error

1.7.3 Binary operator - `A {%,mod} B`

- If `A` and `B` are null, return `(Long)0`
- If `A` or `B` is a `BigDecimal`, `Float`, `Double`, or `String` containing `.`, `e`, or `E`, coerce both `A` and `B` to `Double` and apply operator
- If `A` or `B` is a `BigInteger`, coerce both to `BigInteger` and return `A.remainder(B)`.
- Otherwise coerce both `A` and `B` to `Long` and apply operator
- If operator results in exception, error

1.7.4 Unary minus operator - `-A`

- If `A` is null, return `(Long)0`
- If `A` is a `BigDecimal` or `BigInteger`, return `A.negate()`.
- If `A` is a `String`:
  - If `A` contains `.`, `e`, or `E`, coerce to a `Double` and apply operator
  - Otherwise, coerce to a `Long` and apply operator
  - If operator results in exception, error
- If `A` is `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`
  - Retain type, apply operator
  - If operator results in exception, error
- Otherwise, error

1.8 String Concatenation Operator - `A += B`

To evaluate `A += B`

- Coerce `A` and `B` to `String`.
- Return the concatenated string of `A` and `B`.

1.9 Relational Operators

The relational operators are:

- `==` and `eq`
- `!=` and `ne`
- `<` and `lt`
- `>` and `gt`
- `<=` and `le`
- `>=` and `ge`

The second versions of the last 4 operators are made available to avoid having to use entity references in XML syntax and have the exact same behavior, i.e. `<` behaves the same as `lt` and so on.

The evaluation of relational operators is described in the following sections.

1.9.1 `A {<,>,<=,>=,lt,gt,le,ge} B`

- If `A==B`, if operator is `<=`, `le`, `>=`, or `ge` return `true`.
- If `A` is null or `B` is null, return `false`
- If `A` or `B` is `BigDecimal`, coerce both `A` and `B` to `BigDecimal` and use the return value of `A.compareTo(B)`.
- If `A` or `B` is `Float` or `Double` coerce both `A` and `B` to `Double` apply operator
- If `A` or `B` is `BigInteger`, coerce both `A` and `B` to `BigInteger` and use the return value of `A.compareTo(B)`.
- If `A` or `B` is `Byte`, `Short`, `Character`, `Integer`, or `Long` coerce both `A` and `B` to `Long` and apply operator
- If `A` or `B` is `String` coerce both `A` and `B` to `String`, compare lexically
- If `A` is `Comparable`, then:
  - If `A.compareTo(B)` throws exception, error.
  - Otherwise use result of `A.compareTo(B)`
- If `B` is `Comparable`, then:
  - If `B.compareTo(A)` throws exception, error.
  - Otherwise use result of `B.compareTo(A)`
- Otherwise, error

1.9.2 `A {==,!=,eq,ne} B`

- If `A==B`, apply operator
- If `A` is null or `B` is null return `false` for `==` or `eq`, `true` for `!=` or `ne`.

- If `A` or `B` is `BigDecimal`, coerce both `A` and `B` to `BigDecimal` and then:
  - If operator is `==` or `eq`, return `A.equals(B)`
  - If operator is `!=` or `ne`, return `!A.equals(B)`
- If `A` or `B` is `Float` or `Double` coerce both `A` and `B` to `Double`, apply operator
- If `A` or `B` is `BigInteger`, coerce both `A` and `B` to `BigInteger` and then:
  - If operator is `==` or `eq`, return `A.equals(B)`
  - If operator is `!=` or `ne`, return `!A.equals(B)`
- If `A` or `B` is `Byte`, `Short`, `Character`, `Integer`, or `Long` coerce both `A` and `B` to `Long`, apply operator
- If `A` or `B` is `Boolean` coerce both `A` and `B` to `Boolean`, apply operator
- If A or B is an enum, coerce both A and B to enum, apply operator
- If `A` or `B` is `String` coerce both `A` and `B` to `String`, compare lexically
- Otherwise if an error occurs while calling `A.equals(B)`, error
- Otherwise, apply operator to result of `A.equals(B)`

1.10 Logical Operators

The logical operators are:

- `&&` and `and`
- `||` and `or`
- `!` and `not`

The evaluation of logical operators is described in the following sections.

1.10.1 Binary operator - `A {&&,||,and,or} B`

- Coerce both `A` and `B` to `Boolean`, apply operator

The operator stops as soon as the expression can be determined, i.e., `A and B and C and D` – if `B` is false, then only `A and B` is evaluated.

1.10.2 Unary not operator - `{!,not} A`

- Coerce `A` to `Boolean`, apply operator.

1.11 Empty Operator - `empty A`

The `empty` operator is a prefix operator that can be used to determine if a value is null or empty.

To evaluate `empty A`

- If `A` is null, return `true`
- Otherwise, if `A` is the empty string, then return `true`
- Otherwise, if `A` is an empty array, then return `true`
- Otherwise, if `A` is an empty `Map`, return `true`
- Otherwise, if `A` is an empty `Collection`, return `true`
- Otherwise return `false`

1.12 Conditional Operator - `A ? B : C`

Evaluate `B` or `C`, depending on the result of the evaluation of `A`.

- Coerce `A` to `Boolean`:
  - If `A` is `true`, evaluate and return `B`
  - If `A` is `false`, evaluate and return `C`

1.13 Assignment Operator - `A = B`

Assign the value of `B` to `A`. `A` must be a *lvalue*, otherwise, a `PropertyNotWritableException` will be thrown.

The assignment operator is right-associative. For instance, `A=B=C` is the same as `A=(B=C)`.

To evaluate `expr-a = expr-b`,

- Evaluate `expr-a`, up to the last property resolution, to (`base-a, prop-a`)
- If `base-a` is `null`, and `prop-a` is a `String`,
  - If `prop-a` is a Lambda parameter, throw a `PropertyNotWritableException`

- If `prop-a` is an EL variable (see Section 1.19), evaluate the `ValueExpression` the variable was set to, to obtain the new (`base-a, prop-a`)

- Evaluate `expr-b`, to `value-b`
- Invoke `ELResolver.setValue(base-a, prop-a, value-b)`
- Return `value-b`

The behavior of the assignment operator is determined by the `ELResolver`. For instance, in a stand-alone environment, the class `StandardELContext` contains a default `ELResolver` that allows the assignment of an expression to a non-existing name, resulting in the creation of a bean with the given name in the local bean repository. A JSP container may use the ScopeAttributeELResolver to assign values to scope attributes, or to create attributes in the page scope.

1.14 Semicolon Operator - `A ; B`

The semicolon operator behaves like the comma operator in C.

To evaluate `A;B`, `A` is first evaluated, and its value is discarded. `B` is then evaluated and its value is returned.

1.15 Parentheses

Parentheses can be used to change precedence, as in: `${(a*(b+c))}`

1.16 Operator Precedence

Highest to lowest, left-to-right.

- `[] .`
- `()`
- `-` (unary) `not ! empty`
- `* / div % mod`
- `+ - (binary)`
- `+=`

- `< > <= >= lt gt le ge`
- `== != eq ne`
- `&& and`
- `|| or`
- `? :`
- `->` (Lambda Expression)
- `=`
- `;`

Qualified functions with a namespace prefix have precedence over the operators. Thus the expression `${c?b:f()}` is illegal because `b:f()` is being parsed as a qualified function instead of part of a conditional expression. As usual, `()` can be used to make the precedence explicit, e.g `${c?b:(f())}`.

The symbol -> in a Lambda Expression behaves like an operator for the purpose of ordering the operator precedence, and it has a higher precedence than the assignment and semicolon operators. The following examples illustrates when () is and is not needed.

```
v = x->x+1

x-> (a=x)

x-> c?x+1:x+2
```

All operators are left associative except for the `?:`, `=`, and `->` operators, which are right associative. For instance, `a=b=c` is the parsed as `a=(b=c)`, and `x->y->x+y` is parsed as `x->(y->x+y)`.

1.17 Reserved Words

The following words are reserved for the language and must not be used as identifiers.

```
and    eq     gt     true    instanceof
or     ne     le     false   empty
not    lt     ge     null    div

mod
```

Note that many of these words are not in the language now, but they may be in the future, so developers must avoid using these words.

1.18 Functions

The EL has qualified functions, reusing the notion of qualification from XML namespaces (and attributes), XSL functions, and JSP custom actions. Functions are mapped to public static methods in Java classes.

The full syntax is that of qualified n-ary functions:

$[ns:]f([a_1[,a_2[,...[,a_n]]]])$

Where `ns` is the namespace prefix, `f` is the name of the function, and `a` is an argument.

EL functions are mapped, resolved and bound at parse time. It is the responsibility of the `FunctionMapper` class to provide the mapping of namespace-qualified functions to static methods of specific classes when expressions are created. If no `FunctionMapper` is provided (by passing in `null`), functions are disabled.

1.19 Variables

Just like `FunctionMapper` provides a flexible mechanism to add functions to the EL, `VariableMapper` provides a flexible mechanism to support the notion of EL variables. An EL variable does not directly refer to a model object that can then be resolved by an `ELResolver`. Instead, an EL variable refers to an EL expression. The evaluation of that EL expression yields the value associated with the EL variable.

EL variables are mapped, resolved and bound at parse time. It is the responsibility of the `VariableMapper` class to provide the mapping of EL variables to `ValueExpressions` when expressions are created. If no `VariableMapper` is provided (by passing in `null`), variable mapping is disabled.

See the `javax.el` package description for more details.

1.20 Lambda Expressions

A lambda expression is a `ValueExpression` with parameters. The syntax is similar to the lambda expression in the Java Language, except that in EL, the body of the lambda expression is an EL expression. These are some examples:

- `x->x+1`
- `(x,y)->x+y`
- `()->64`

The identifiers to the left of -> are lambda parameters. The parenthesis is optional if and only if there is one parameter.

A lambda expression behaves like a function. It can be invoked immediately,

- `((x,y)->x+y)(3,4)` evaluates to 7.

When a lambda expression is assigned, it can be referenced and invoked indirectly,

- `v = (x,y)->x+y; v(3,4) evaluates to 7`
- `fact = n -> n==0? 1: n*fact(n-1); fact(5)` evaluates to 120

It can also be passed as an argument to a method, and be invoked in the method, by invoking `javax.el.LambdaExpression.invoke()`, such as

- `employees.where(e->e.firstName == ‘Bob’)`

When a lambda expression is invoked, the expression in the body is evaluated, with its formal parameters replaced by the arguments supplied at the invocation. The number of arguments must be equal to or more than the number the formal parameters. Any extra arguments are ignored.

A lambda expression can be nested within another lambda expression, like

- `customers.select(c->[c.name, c.orders.sum(o->o.total)])`

The scope of a lambda argument is the body of the lambda expression. A lambda argument hides other EL variables, identifiers or arguments of the nesting lambda expressions, of the same name.

Note that in the case of nested lambda expressions where the body of the inner lambda expression contains references to parameters of outer lambda expressions, such as

- `x->y->x+y`

the scope of the outer lambda parameters extends to cover the inner body. For instance, with the above example, the argument *x* must be in scope when *x+y* is evaluated, even though the body of the outer lambda expression has already been executed.

1.21 Enums

The Unified EL supports Java SE 5 enumerated types. Coercion rules for dealing with enumerated types are included in the following section. Also, when referring to values that are instances of an enumerated type from within an EL expression, use the literal string value to cause coercion to happen via the below rules. For example, Let's say we have an enum called `Suit` that has members `Heart`, `Diamond`, `Club`, and `Spade`. Furthermore, let's say we have a reference in the EL, `mySuit`, that is a `Spade`. If you want to test for equality with the `Spade` enum, you would say `${mySuit == 'Spade'}`. The type of the mySuit will trigger the invocation of `Enum.valueOf(Suit.class, 'Spade')`.

1.22 Static Field and Method Reference

A static field or static method of a Java class can be referenced with the syntax *classname.field*, such as

```
Boolean.TRUE
```

the classname is the name of a class, without the package name.

An enum constant is a public static field, so the same syntax can be used to refer to an enum constant, like the following:

```
RoundingMode.FLOOR
```

1.22.1 Access Restrictions and Imports

For security, the following restrictions are enforced.

1. Only the public static fields and methods can be referenced.
2. Static fields cannot be modified.
3. Except for classes with `java.lang.*` package names, a class has to be explicitly imported before its static fields or methods can be referenced.

1.22.2 Imports of Packages, Classes, and Static Fields

Either a class or a package can be explicitly imported into the EL evaluation environment. Importing a package imports all the classes in the package. The classes that can be imported are restricted to the classes that can be loaded by the current class loader.

By default, the following packages are imported by the EL environment.

```
java.lang.*
```

A static field can also be imported statically. A statically imported static field can be referenced by the field name, without the classname.

The imports of packages, classes, and static fields are handled by the `ImportHandler` in the `ELContext`.

1.22.3 Constructor Reference

A class name reference, followed by arguments in parenthesis, such as

```
Boolean(true)
```

denotes the invocation of the constructor of the class with the supplied arguments. The same restrictions (the class must be public and has already been imported) for static methods apply to the constructor calls.

1.23 Type Conversion

Every expression is evaluated in the context of an expected type. The result of the expression evaluation may not match the expected type exactly, so the rules described in the following sections are applied.

Custom type conversions can be specified in an `ELResolver` by implementing the method `convertToType`. More than one `ELResolvers` can be specified for performing custom conversions, and they are selected and applied in the order of their positions in the `ELResolver` chain, as usual.

During expression evaluations, the custom type converters are first selected and applied. If there is no custom type converter for the conversion, the default conversions specified in the following sections are used.

1.23.1 To Coerce a Value X to Type Y

- If `X` is `null` and `Y` is not a primitive type and also not a `String`, return `null`.
- If `X` is of a primitive type, Let `X'` be the equivalent "boxed form" of `X`. Otherwise, Let `X'` be the same as `X`.
- If `Y` is of a primitive type, Let `Y'` be the equivalent "boxed form" of `Y`. Otherwise, Let `Y'` be the same as `Y`.
- Apply the rules in Sections 1.23.2-1.23.7 for coercing `X'` to `Y'`.
- If `Y` is a primitive type, then the result is found by "unboxing" the result of the coercion. If the result of the coercion is `null`, then error.
- If `Y` is not a primitive type, then the result is the result of the coercion.

For example, if coercing an `int` to a `String`, "box" the `int` into an `Integer` and apply the rule for coercing an `Integer` to a `String`. Or if coercing a `String` to a `double`, apply the rule for coercing a `String` to a `Double`, then "unbox" the resulting `Double`, making sure the resulting `Double` isn't actually `null`.

1.23.2 Coerce A to String

- If `A` is `null`: return `""`
- Otherwise, if `A` is `String`: return `A`
- Otherwise, if `A` is `Enum`, return `A.name()`
- Otherwise, if `A.toString()` throws an exception, error
- Otherwise, return `A.toString()`

1.23.3 Coerce A to Number type N

- If `A` is `null` and `N` is not a primitive type, return `null`.
- If `A` is `null` or `""`, return 0.
- If `A` is `Character`, convert `A` to `new Short((short)a.charValue())`, and apply the following rules.
- If `A` is `Boolean`, then error.
- If `A` is `Number` type `N`, return A
- If `A` is `Number`, coerce quietly to type `N` using the following algorithm:
  - If `N` is `BigInteger`
    - If `A` is a `BigDecimal`, return `A.toBigInteger()`
    - Otherwise, return `BigInteger.valueOf(A.longValue())`
  - If `N` is `BigDecimal`,

        - If `A` is a `BigInteger`, return `new BigDecimal(A)`
        - Otherwise, return `new BigDecimal(A.doubleValue())`
    - If `N` is `Byte`, return `new Byte(A.byteValue())`
    - If `N` is `Short`, return `new Short(A.shortValue())`
    - If `N` is `Integer`, return `new Integer(A.intValue())`
    - If `N` is `Long`, return `new Long(A.longValue())`
    - If `N` is `Float`, return `new Float(A.floatValue())`
    - If `N` is `Double`, return `new Double(A.doubleValue())`
    - Otherwise, error.
- If `A` is `String`, then:
    - If `N` is `BigDecimal` then:
        - If `new BigDecimal(A)` throws an exception then error.
        - Otherwise, return `new BigDecimal(A)`.
- If `N` is `BigInteger` then:
        - If `new BigInteger(A)` throws an exception then error.
        - Otherwise, return `new BigInteger(A)`.
    - If `N.valueOf(A)` throws an exception, then error.
    - Otherwise, return `N.valueOf(A)`.
- Otherwise, error.

1.23.4 Coerce A to Character or char

- If `A` is `null` and the target type is not the primitive type `char`, return `null`
- If `A` is `null` or `""`, return `(char)0`
- If `A` is `Character`, return `A`
- If `A` is `Boolean`, error
- If `A` is `Number`, coerce quietly to type `Short`, then return a `Character` whose numeric value is equivalent to that of a `Short`.
- If `A` is `String`, return `A.charAt (0)`
- Otherwise, error

1.23.5 Coerce A to Boolean or boolean

- If `A` is `null` and the target type is not the primitive type `boolean`, return `null`
- If `A` is `null` or `""`, return `false`

- Otherwise, if `A` is a `Boolean`, return `A`
- Otherwise, if `A` is a `String`, and `Boolean.valueOf(A)` does not throw an exception, return it
- Otherwise, error

1.23.6 Coerce A to an Enum Type T

- If `A` is `null`, return `null`
- If `A` is assignable to `T`, coerce quietly
- If `A` is `""`, return `null`.
- If `A` is a `String` call `Enum.valueOf(T.getClass(), A)` and return the result.

1.23.7 Coerce A to Any Other Type T

- If `A` is `null`, return `null`
- If `A` is assignable to `T`, coerce quietly
- If `A` is a `String`, and `T` has no `PropertyEditor`:
  - If `A` is `""`, return `null`
  - Otherwise error
- If `A` is a `String` and `T`'s `PropertyEditor` throws an exception:
  - If `A` is `""`, return `null`
  - Otherwise, error
- Otherwise, apply `T`'s `PropertyEditor`
- Otherwise, error

1.24 Collected Syntax

The following is a javaCC grammar with syntax tree generation. It is meant to be used as a guide and reference only.

```
/* == Option Declaration == */
options
```

```
{
    STATIC=false;
    NODE_PREFIX="Ast";
    VISITOR_EXCEPTION="javax.el.ELException";
    VISITOR=false;
    MULTI=true;
    NODE_DEFAULT_VOID=true;
    JAVA_UNICODE_ESCAPE=false;
    UNICODE_INPUT=true;
    BUILD_NODE_FILES=true;
}
/* == Parser Declaration == */
PARSER_BEGIN( ELParser )
package com.sun.el.parser;
import java.io.StringReader;
import javax.el.ELException;
public class ELParser
{
    public static Node parse(String ref) throws ELException
    {
        try {
            return (new ELParser(new
StringReader(ref))).CompositeExpression();
        } catch (ParseException pe) {
            throw new ELException(pe.getMessage());
        }
    }
}
PARSER_END( ELParser )
/*
 * CompositeExpression
 * Allow most flexible parsing, restrict by examining
 * type of returned node
 */
AstCompositeExpression CompositeExpression() #CompositeExpression :
{}
```

```
{
    (DeferredExpression() |
     DynamicExpression() |
     LiteralExpression())* <EOF> { return jjtThis; }
}

/*
 * LiteralExpression
 * Non-EL Expression blocks
 */
void LiteralExpression() #LiteralExpression : { Token t = null; }
{
    t=<LITERAL_EXPRESSION> { jjtThis.setImage(t.image); }
}

/*
 * DeferredExpression
 * #{..} Expressions
 */
void DeferredExpression() #DeferredExpression : {}
{
    <START_DEFERRED_EXPRESSION> Expression() <RCURL>
}

/*
 * DynamicExpression
 * ${..} Expressions
 */
void DynamicExpression() #DynamicExpression : {}
{
    <START_DYNAMIC_EXPRESSION> Expression() <RCURL>
}
/*
 * Expression
 * EL Expression Language Root
 */
```

```
void Expression() : {}
{
    SemiColon()
}

/*
 * SemiColon
 */
void SemiColon() : {}
{
    Assignment() (<SEMICOLON> Assignment() #SemiColon(2) )*
}

/*
 * Assignment
 * For '=', right associatve, then LambdaExpression or Choice or
Assignment
 */
void Assignment() : {}
{
    LOOKAHEAD(3) LambdaExpression() |
    Choice() (<ASSIGN> Assignment() #Assign(2) )?
}

/*
 * LambdaExpression
 */
void LambdaExpression() #LambdaExpression : {}
{
    LambdaParameters() <ARROW>
    (LOOKAHEAD(3) LambdaExpression() | Choice() )
}

void LambdaParameters() #LambdaParameters: {}
{
    Identifier()
    | <LPAREN (Identifier() (<COMMA> Identifier())*)? <RPAREN>
}

/*
 * Choice
 * For Choice markup a ? b : c, right associative
 */
void Choice() : {}
{
    Or() (<QUESTIONMARK> Choice() <COLON> Choice() #Choice(3))?
}

/*
 * Or
 * For 'or' '||', then And
 */
void Or() : {}
{
    And() ((<OR0>|<OR1>) And() #Or(2))*
}

/*
 * And
 * For 'and' '&&', then Equality
 */
void And() : {}
{
    Equality() ((<AND0>|<AND1>) Equality() #And(2))*
}
/*
 * Equality
 * For '==' 'eq' '!=' 'ne', then Compare
 */
void Equality() : {}
{
    Compare()
    (
        ((<EQ0>|<EQ1>) Compare() #Equal(2))
    |
        ((<NE0>|<NE1>) Compare() #NotEqual(2))
    )*
}

/*
 * Compare
 * For a bunch of them, then Math
 */
void Compare() : {}
{
    Concatenation()
    (
        ((<LT0>|<LT1>) Concatenation() #LessThan(2))
    |
        ((<GT0>|<GT1>) Concatenation() #GreaterThan(2))
    |
        ((<LE0>|<LE1>) Concatenation() #LessThanEqual(2))
    |
        ((<GE0>|<GE1>) Concatenation() #GreaterThanEqual(2))
    )*
}
/*
 * Concatenation
 * For 'cat', then Math()
 */
void Concatenation() : {}
{
    Math() ( <CONCAT> Math() #Concat(2) )*
}

/*
 * Math
 * For '+' '-', then Multiplication
 */
void Math() : {}
{
    Multiplication()
    (
        (<PLUS> Multiplication() #Plus(2))
    |
        (<MINUS> Multiplication() #Minus(2))
    )*
}
/*
 * Multiplication
 * For a bunch of them, then Unary
 */
void Multiplication() : {}
{
    Unary()
    (
        (<MULT> Unary() #Mult(2))
    |
        ((<DIV0>|<DIV1>) Unary() #Div(2))
    |
        ((<MOD0>|<MOD1>) Unary() #Mod(2))
    )*
}

/*
 * Unary
 * For '-' '!' 'not' 'empty', then Value
 */
void Unary() : {}
{
        <MINUS> Unary() #Negative
    |
        (<NOT0>|<NOT1>) Unary() #Not
    |
        <EMPTY> Unary() #Empty
    |
        Value()
}
/*
 * Value
 * Defines Prefix plus zero or more Suffixes
 */
void Value() : {}
{
    (ValuePrefix() (ValueSuffix())*) #Value(>1)
}

/*
 * ValuePrefix
 * For Literals, Variables, and Functions
 */
void ValuePrefix() : {}
{
    Literal() | NonLiteral()
}

/*
 * ValueSuffix
 * Either dot or bracket notation
 */
void ValueSuffix() : {}
{
    DotSuffix() | BracketSuffix()
}

/*
 * DotSuffix
 * Dot Property and Dot Method
 */
void DotSuffix() #DotSuffix : {  Token t = null; }
```

```
{
    <DOT> t=<IDENTIFIER> { jjtThis.setImage(t.image); }
    (MethodArguments())?
}
/*
 * BracketSuffix
 * Sub Expression Suffix
 */
void BracketSuffix() #BracketSuffix : {}
{
        <LBRACK> Expression() <RBRACK>
        (MethodArguments())?
}
/*
 * MethodArguments
 */
void MethodArguments() #MethodArguments : {}
{
    <LPAREN> (Expression() (<COMMA> Expression())*)? <RPAREN>
}

/*
 * Parenthesized Lambda Expression, with optional invokation
 */
void LambdaExpressionOrCall() #LambdaExpression : {}

{
    <LPAREN>
        LambdaParameters() <ARROW>
        (LOOKAHEAD(3) LambdaExpression() | Choice() )
    <RPAREN>
    (MethodArguments())*
}
/*
 * NonLiteral
 * For Grouped Operations, Identifiers, and Functions
 */
void NonLiteral() : {}
{
    LOOKAHEAD(4) LambdaExpressionOrCall()
    | <LPAREN> Expression() <RPAREN>
    | LOOKAHEAD(4) Function()
    | Identifier()
    | MapData()
    | ListData()
}

void MapData() #MapData: {}
{
    <START_MAP>
        ( MapEntry() ( <COMMA> MapEntry() )* )?
    <RCURL>
}

void MapEntry() #MapEntry: {}
{
    Expression() (<COLON> Expression())?
}

void ListData() #ListData: {}
{
    <LBRACK>
        ( Expression() ( <COMMA> Expression() )* )?
    <RBRACK>
}

/*
 * Identifier
 * Java Language Identifier
 */
void Identifier() #Identifier : { Token t = null; }
{
    t=<IDENTIFIER> { jjtThis.setImage(t.image); }
}
/*
 * Function
 * Namespace:Name(a,b,c)
 */
void Function() #Function :
{
    Token t0 = null;
    Token t1 = null;
}
{
    t0=<IDENTIFIER> (<COLON> t1=<IDENTIFIER>)?
    {
        if (t1 != null) {
            jjtThis.setPrefix(t0.image);
            jjtThis.setLocalName(t1.image);
        } else {
            jjtThis.setLocalName(t0.image);
        }
    }
    (MethodArguments())+
}
/*
 * Literal
 * Reserved Keywords
 */
void Literal() : {}
{
    Boolean()
    | FloatingPoint()
    | Integer()
    | String()
    | Null()
}
/*
```

```
 * Boolean
 * For 'true' 'false'
 */
void Boolean() : {}
{
    <TRUE> #True
    | <FALSE> #False
}
/*
 * FloatinPoint
 * For Decimal and Floating Point Literals
 */
void FloatingPoint() #FloatingPoint : { Token t = null; }
{
    t=<FLOATING_POINT_LITERAL> { jjtThis.setImage(t.image); }
}
/*
 * Integer
 * For Simple Numeric Literals
 */
void Integer() #Integer : { Token t = null; }
{
    t=<INTEGER_LITERAL> { jjtThis.setImage(t.image); }
}
/*
 * String
 * For Quoted Literals
 */
void String() #String : { Token t = null; }
{
    t=<STRING_LITERAL> { jjtThis.setImage(t.image); }
}
/*
 * Null
 * For 'null'
 */
```

```
void Null() #Null : {}
{
    <NULL>
}
/* =====================================================================
========= */TOKEN_MGR_DECLS:
{
    java.util.Stack<Integer> stack = new java.util.Stack<Integer>();
}

<DEFAULT> TOKEN :
{
  < LITERAL_EXPRESSION:
    ((~["\\", "$", "#"])
      | ("\\" ("\\" | "$" | "#"))
      | ("$" ~["{", "$"])
      | ("#" ~["{", "#"])
    )+
    | "$"
    | "#"
  >
|
  < START_DYNAMIC_EXPRESSION: "${" > {stack.push(DEFAULT);}:
IN_EXPRESSION
|
  < START_DEFERRED_EXPRESSION: "#{" > {stack.push(DEFAULT);}:
IN_EXPRESSION
}

<DEFAULT> SKIP : { "\\" }

<IN_EXPRESSION, IN_MAP> SKIP:
{ " " | "\t" | "\n" | "\r" }

<IN_EXPRESSION, IN_MAP> TOKEN :
{
        < START_MAP : "{" > {stack.push(curLexState);}: IN_MAP
|       < RCURL: "}" > {SwitchTo(stack.pop());}
|       < INTEGER_LITERAL: ["0"-"9"] (["0"-"9"])* >
|       < FLOATING_POINT_LITERAL: (["0"-"9"])+ "." (["0"-"9"])*
(<EXPONENT>)?
                | "." (["0"-"9"])+ (<EXPONENT>)?
                | (["0"-"9"])+ <EXPONENT>
        >
|       < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
|       < STRING_LITERAL: ("\"" ((~["\"","\\"])
                | ("\\" ( ["\\","\""] )))* "\"")
                | ("\'" ((~["\'","\\"])
                | ("\\" ( ["\\","\'"] )))* "\'")
        >
|       < BADLY_ESCAPED_STRING_LITERAL: ("\"" (~["\"","\\"])* ("\\"
( ~["\\","\""] )))
                | ("\'" (~["\'","\\"])* ("\\" ( ~["\\","\'"] )))
        >
|       < TRUE : "true" >
|       < FALSE : "false" >
|       < NULL : "null" >
|       < DOT : "." >
|       < LPAREN : "(" >
|       < RPAREN : ")" >
|       < LBRACK : "[" >
|       < RBRACK : "]" >
|       < COLON : ":" >
|       < COMMA : "," >
|       < SEMICOLON : ";" >
|       < GT0 : ">" >
|       < GT1 : "gt" >
|       < LT0 : "<" >
|       < LT1 : "lt" >
|       < GE0 : ">=" >
|       < GE1 : "ge" >
|       < LE0 : "<=" >
|       < LE1 : "le" >
|       < EQ0 : "==" >
|       < EQ1 : "eq" >
|       < NE0 : "!=" >
|       < NE1 : "ne" >
|       < NOT0 : "!" >
|       < NOT1 : "not" >
|       < AND0 : "&&" >
|       < AND1 : "and" >
|       < OR0 : "||" >
|       < OR1 : "or" >
|       < EMPTY : "empty" >
|       < INSTANCEOF : "instanceof" >
|       < MULT : "*" >
|       < PLUS : "+" >
|       < MINUS : "-" >
|       < QUESTIONMARK : "?" >
|       < DIV0 : "/" >
|       < DIV1 : "div" >
|       < MOD0 : "%" >
|       < MOD1 : "mod" >
|       < CONCAT : "+=" >
|       < ASSIGN : "=" >
|       < ARROW : "->" >
|    < IDENTIFIER : (<LETTER>|<IMPL_OBJ_START>) (<LETTER>|<DIGIT>)* >
|       < #IMPL_OBJ_START: "#" >
|       < #LETTER:
                [
                "\u0024",
                "\u0041"-"\u005a",
                "\u005f",
                "\u0061"-"\u007a",
                "\u00c0"-"\u00d6",
                "\u00d8"-"\u00f6",
                "\u00f8"-"\u00ff",
                "\u0100"-"\u1fff",
                "\u3040"-"\u318f",
                "\u3300"-"\u337f",
                "\u3400"-"\u3d2d",
                "\u4e00"-"\u9fff",
                "\uf900"-"\ufaff"
                ]
        >
|       < #DIGIT:
                [
                "\u0030"-"\u0039",
                "\u0660"-"\u0669",
                "\u06f0"-"\u06f9",
                "\u0966"-"\u096f",
                "\u09e6"-"\u09ef",
                "\u0a66"-"\u0a6f",
                "\u0ae6"-"\u0aef",
                "\u0b66"-"\u0b6f",
                "\u0be7"-"\u0bef",
                "\u0c66"-"\u0c6f",
                "\u0ce6"-"\u0cef",
                "\u0d66"-"\u0d6f",
                "\u0e50"-"\u0e59",
                "\u0ed0"-"\u0ed9",
                "\u1040"-"\u1049"
                ]
        >
|       < ILLEGAL_CHARACTER: (~[]) >
}
```

Notes

- * = 0 or more, + = 1 or more, ? = 0 or 1.
- An identifier is constrained to be a Java identifier - e.g., no `-`, no `/`, etc.
- A `String` only recognizes a limited set of escape sequences, and \ may not appear unescaped.

- The relational operator for equality is `==` (double equals).
- The value of an `IntegerLiteral` ranges from `Long.MIN_VALUE` to `Long.MAX_VALUE`
- The value of a `FloatingPointLiteral` ranges from `Double.MIN_VALUE` to `Double.MAX_VALUE`
- It is illegal to nest `${` or `#{` inside an outer `${` or `#{`.

CHAPTER 2

Operations on Collection Objects

This chapter describes how collection objects and literals can be constructed in the EL expression, and how collection objects can be manipulated and processed by applying operations in a pipeline.

2.1 Overview

To provide support for collection objects, EL includes syntaxes for constructing sets, lists, and maps dynamically. Any EL expressions, not just literals, can be used in the construction.

EL also includes a set of operations that can be applied on collections. By design, the methods supporting these operations have names and semantics very similar to those in Java SE 8 libraries. Since EL and Java have different syntaxes and capabilities, they are not identical, but they are similar enough that users should have no problem switching from one to the other.

Since the methods supporting the collection operations do not exist in Java SE 7, they are implemented in the Expression Language with `ELResolvers`. In an EL expression, collection operations are carried out by invoking methods, and no special syntaxes are introduced for them. Strictly speaking, these operations are not part of the expression language, and can be taken as examples of what can be achieved with the expression language. The specification specifies the syntaxes and behaviors of a standard set of collection operations. However, an user can easily add, extend and modify the behavior of the operations by providing customized `ELResolvers`.

Compared to Java SE 8, the collection support in EL has a much smaller and simpler scope. Although EL does not disallow collections of infinite size, it works best when the collection objects are created in memory, with known sizes.   It also does not address the performance issue in a multi-threaded environment, and does not provide explicit controls for evaluating collection operations in parallel. The future version of EL will likely include functionalities from Java SE 8, when it is released.

Central to the implementation is the use of lambda expressions, now supported in EL. A lambda expression in the Java language is used to specify a method in an anonymous implementation of a functional interface. The concept of a lambda expression in EL is much simpler: it is just an anonymous function that can be passed as an argument to a method, to be evaluated in the method when needed. In the collection operations, lambda expressions are specified as arguments to the methods supporting the operations. Usually when the lambda expressions are invoked, an element from stream of the collection is passed as an argument to the lambda expression. For instance, the argument to the `filter` method is a lambda expression which acts as a predicate function to determine if an element should be included in the resulting stream.

2.2 Construction of Collection Objects

EL allows the construction of sets, lists, and maps dynamically. Any EL expressions, including nested collection constructions, can be used in the construction. These expressions are evaluated at the time of the construction.

2.2.1 Set Construction

Construct an instance of `java.lang.util.Set<Object>`.

2.2.1.1 Syntax

```
SetData := '{' DataList '}'
DataList := (expression (',' expression)* )?
```

2.2.1.2 Example

```
{1, 2, 3}
```

2.2.2 List Construction

Construct an instance of `java.lang.util.List<Object>`.

2.2.2.1 Syntax

```
ListData := ‘[‘ DataList ‘]’
DataList := (expression (‘,’ expression)* )?
```

2.2.2.2 Example

```
[1, "two", [foo, bar]]
```

2.2.3 Map Construction

Construct an instance of `java.lang.util.Map<Object>`.

2.2.3.1 Syntax

```
Map := '{' MapEntries '}'
MapEntries := (MapEntry (',' MapEntry)* )?
MapEntry := expression ':' expression
```

2.2.3.2 Example

```
{"one":1, "two":2, "three":3}
```

2.3 Collection Operations

2.3.1 Stream and Pipeline

The operations on a collection object are realized as method calls to the stream of elements derived from the collection. The method `stream` can be used to obtain a `Stream` from a `java.util.Collection` or a Java array.

To obtain a `Stream` from a `Map`, the collection view of a `Map`, such as `MapEntry` can be used as the source of `Stream`.

Some operations return another `Stream`, which allows other operations. Therefore the operations can be chained together to form a pipeline. For example, to get a list of titles of history books, one can write in EL:

```
books.stream().filter(b->b.category == 'history')
              .map(b->b.title)
              .toList()
```

A stream pipeline consists of

- The source,
- Intermediate operations, and
- A terminal operation.

The source of a pipeline is the `Stream` object.

An intermediate operation is a method in `Stream` that returns a `Stream`. A pipeline may contain zero or more intermediate operations.

A pipeline ends in a terminal operation. A terminal operation is a method in `Stream` that does not return a `Stream`.

The execution of a pipeline only begins when the terminal operation starts its execution. Most of the intermediate operations are evaluated lazily: they only yield as many elements in the stream as are required by the downstream operations. Because of this, they need not keep intermediate results of the operations. For instance, the `filter` operation does not keep a collection of the filtered elements.

A notable exception is the `sorted` operation, since all elements are needed for sorting.

The specification specifies the behavior of the operations in a pipeline, and does not specify the implementation of a pipeline. The operations must not modify the source collection. The user must also make sure that the source collection is not modified externally during the execution of the pipeline, otherwise the behavior of the collection operations will be undefined.

The behavior of the operations are undefined if the collection contains null elements. Null elements in a collection should be removed by a filter to obtain consistent results.

The source stream in a pipeline that has already started its execution cannot be used in another pipeline, otherwise the behavior is undefined.

2.3.2 Operation Syntax Description

The implementation of `Stream` that contains the methods supporting the operations are not part of the API. The syntax and the behavior of the operations are described in this chapter.

For documentation purposes, pseudo method declarations are used in this chapter for the operations. A method includes

- The return type
- The type of the source stream
- The method name
- The method parameters

A typical method declaration would looks like

```
returnT Stream<T>.method(T1 arg1, T2 arg2)
```

Some methods have optional parameters. The declarations of the methods with all possible combinations of the parameters are listed in the syntax sections, as if they are overloaded. Any `null` parameter will result in a `NullPointerException` at run-time.

Some of the parameters are lambda expressions, also known as functions. A lambda expression can have its own parameters and can return a value. To describe the parameter types and the return type of a lambda expression, the following is an example of the notation that is used.

```
(p1,p2)->returnT
```

For instance, the declaration for the operation `filter` is

```
Stream<S> Stream<S>.filter((S->boolean) predicate)
```

From this we know that the source object is a `Stream` of `S`, and the return object is also a `Stream`, of the same type. The operator takes a `predicate` function (lambda expression) as an argument. The argument of the function is an element of the source, and the function returns a `boolean`.

The generic types in the declaration are used only to help the readers to identify the type relationships among various parts of the declaration, and do not have the same meaning as used in the Java language. At runtime, EL deals with Objects, and does not track generic types.

2.3.3 Implementation Classes

The specification makes references to some implementation classes that are not in the API. They contains methods whose behaviors are specified in this section.

2.3.3.1 Stream

An instant of `Stream` is obtained by calling the method `stream()` of a `java.util.Collection` object or a Java array. The methods in this class support the stream operations and are described in Section 2.3.5 to Section 2.3.26.

2.3.3.2 Optional

An `Optional` is used to represent a value that may not exist. Instead of using `null` as a default value, the use of `Optional` allows the user to specify a default.

A non-existing or empty value is represented by an empty `Optional`.

An `Optional` is usually the result of a computation over the elements of a `Stream`, where an empty `Stream` results in an empty `Optional`. See for example, Section 2.3.18, "max" on page 2-58.

The following are methods in `Optional<T>`.

- `T get()`

Returns the value held by the `Optional`, or throws an `ELException` if the `Optional` is empty.

- `void ifPresent((x)->void)consumer)`

The value held by the `Optional` is processed by the function `consumer` if it is not empty. See also Section 2.3.4.4 "consumer" on page 2-49.

- `T orElse(T other)`

Returns the value held by the `Optional`, or the value `other` if the `Optional` is empty.

- `T orElseGet((()->T) other)`

Returns the value held by the `Optional`, or the value returned by the lambda expression `other` if the `Optional` is empty.

2.3.4 Functions

Some operations takes functions (lambda expressions) as parameters. Again, we used the notion

```
(arg1Type, ...)->returnType
```

to describe the argument types and the return type of a function.

2.3.4.1 predicate

- `S -> boolean`

This function takes the input argument, usually the element of the source stream, and determines if it satisfies some criteria.

2.3.4.2 mapper

- `S -> R`

This function maps, or transforms the input argument, usually the element of the source stream, to the result.

2.3.4.3 comparator

- `(S, S) -> int`

This function compares two arguments, usually the elements of the source stream, and returns a negative integer, zero, or a positive integer, if the first argument is respectively less than, equal to, or greater than the second argument.

2.3.4.4 consumer

- `S -> void`

This function processes the input argument, usually the element of the source stream, and returns nothing.

2.3.4.5 binaryOperator

- `(S, S) -> S`

This function applies a binary operation to the input arguments, and returns the result. The first argument is usually an internal accumulator value, and the second argument is usually the element of the source stream.

The arguments and the result are of the same type.

2.3.5 filter

2.3.5.1 Syntax

```
Stream<S> Stream<S>.filter((S->boolean) predicate)
```

2.3.5.2 Description

This method produces a stream containing the source stream elements for which the `predicate` function returns `true`. The argument of `predicate` function represents the element to test.

2.3.5.3 See

Section 2.3.4.1 “predicate” on page 2-49

2.3.5.4 Example

To find the products whose price is greater than or equal to 10:

```
    products.stream().filter(p->p.unitPrice >= 10).toList()
```

2.3.6 map

2.3.6.1 Syntax

```
Stream<R> Stream<S>.map((S->R) mapper)
```

2.3.6.2 Description

This method produces a stream by applying the `mapper` function to the elements of the source stream. The argument of `mapper` function represents the element to process, and the result of the `mapper` function represents the element of the resulting `Stream`.

2.3.6.3 See

Section 2.3.4.2 “mapper” on page 2-49

2.3.6.4 Examples

- To get the list of the names of all products:

  ```
  products.stream().map(p->p.name).toList()
  ```

- To creates a list of product names and prices for products with a price greater than or equal to 10:

  ```
  products.stream().filter(p->p.unitPrice >= 10).
                   .map(p->[p.name, p.unitPrice])
                   .toList()
  ```

2.3.7 flatMap

2.3.7.1 Syntax

```
Stream<R> Stream<S>.flatMap((S->Stream<R>) mapper)
```

2.3.7.2 Description

This method produces a stream by mapping each of the source elements to another stream and then concatenating the mapped streams. If the `mapper` function does not return a `Stream`, the behavior is undefined.

2.3.7.3 See

Section 2.3.4.2 “mapper” on page 2-49

2.3.7.4 Examples

- To list all orders of US customers:

  ```
  customers.stream().filter(c->c.country == 'USA')
                    .flatMap(c->c.orders.stream())
                    .toList()
  ```

- To obtain a list of alphabets used in a list of words:

```
words.stream().flatMap(w->w.toCharArray().stream())
              .sorted()
              .distinct()
              .toList()
```

2.3.8 distinct

2.3.8.1 Syntax

`Stream<S> Stream<S>.distinct()`

2.3.8.2 Description

This method produces a stream containing the elements of the source stream that are distinct, according to `Object.equals`.

2.3.8.3 Example

To remove the duplicate element b:

```
['a', 'b', 'b', 'c'].stream().distinct().toArray()
```

2.3.9 sorted

2.3.9.1 Syntax

```
Stream<S> Stream<S>.sorted()
Stream<S> Stream<S>.sorted(((p,q)->int) comparator)
```

2.3.9.2 Description

This method produces a stream containing the elements of the source stream in sorted order. If no `comparator` is specified, the elements are sorted in natural order. The behavior is undefined if no `comparator` is specified, and the elements do not implement `java.lang.Comparable`. If a `comparator` is specified, the elements are sorted with the provided comparator.

The source collection is unaffected by this operation.

2.3.9.3 See

Section 2.3.4.3 “comparator” on page 2-49

2.3.9.4 Examples

- To sort a list of integers

  ```
  [1,3,2,4].stream().sorted().toList()
  ```

- To sort a list of integers in reversed order

  ```
  [1,3,2,4].stream().sorted((i,j)->j-i).List()
  ```

- To sort a list of words in the order of word length; and then for words of the same length, in alphabetical order:

  ```
  words.stream().sorted(
      (s,t)->(s.length()==t.length()? s.compareTo(t)
                                    : s.length() - t.length()))
                .toLst()
  ```

- To sort the products by name:

  ```
  products.stream().sorted(
      (p,q)->p.name.compareTo(p.name)).toList()
  ```

Or by defining a *comparing* function, this can be rewritten as:

```
comparing = map->(x,y)->map(x).compareTo(map(y));
products.stream().sorted(comparing(p->p.name)).toList()
```

2.3.10 forEach

2.3.10.1 Syntax

```
Object stream<S>.forEach(((S)->void)consumer)
```

2.3.10.2 Description

This method invokes the `consumer` function for each element in the source stream.

This method always returns `null`.

2.3.10.3 See

Section 2.3.4.4 "consumer" on page 2-49

2.3.10.4 Example

To print a list of customer names:

```
customers.stream().forEach(c->printer.print(c.name))
```

2.3.11 peek

2.3.11.1 Syntax

```
Stream<S> Stream<S>.peek(((S)->void)consumer)
```

2.3.11.2 Description

This method produces a stream containing the elements of the source stream, and invokes the `consumer` function for each element in the stream. The primary purpose of this method is for debugging, where one can take a peek at the elements in the stream at the place where this method is inserted.

2.3.11.3 See

Section 2.3.4.4 "consumer" on page 2-49

2.3.11.4 Example

To print the a list of integer before and after a filter:

```
[1,2,3,4,5].stream().peek(i->print(i))
                    .filter(i-> i%2 == 0)
                    .peek(i->print(i))
                    .toList()
```

2.3.12 iterator

2.3.12.1 Syntax

```
Iterator<S> Stream<S>.iterator()
```

2.3.12.2 Description

This method returns an iterator for the source stream, suitable for use in Java codes.

2.3.13 limit

2.3.13.1 Syntax

```
Stream<S> Stream<S>.limit(Number count)
```

2.3.13.2 Description

This method produces a stream containing the first `count` number of elements of the source stream.

If `count` is greater than the number of source elements, all the elements are included in the returned stream. If the `count` is less than or equal to zero, an empty stream is returned.

2.3.13.3 Example

To list the 3 most expensive products:

```
    products.stream().sorted(p->p.unitPrice)
                     .limit(3)
                     .toList()
```

2.3.14 substream

2.3.14.1 Syntax

```
Stream<S> Stream<S>.substream(Number start)
Stream<S> Stream<S>.substream(Number start, Number end)
```

2.3.14.2 Description

This method produces a stream containing the source elements, skipping the first `start` elements, and including the rest of the elements in the stream if `end` is not specified, or the next (`end - start`) elements in the stream if `end` is specified.

If the elements in the source stream has fewer than `start` elements, nothing is included. If `start` is less than or equal to zero, no elements are skipped.

2.3.14.3 Example

The example

```
    [1,2,3,4,5].stream().substream(2,4).toArray()
```

produces the array [3,4].

2.3.15 toArray

2.3.15.1 Syntax

```
S[] Stream<S>.toArray()
```

2.3.15.2 Description

This method returns an array containing the elements of the source stream.

2.3.16 toList

2.3.16.1 Syntax

```
List Stream<S>.toList()
```

2.3.16.2 Description

This method returns a `List` containing the elements of the source stream.

2.3.17 reduce

2.3.17.1 Syntax

```
Optional<S> Stream<S>.reduce(((S,S)->S) binaryOperator)
S Stream<S>.reduce(S seed, ((S,S)->S) binaryOperator))
```

2.3.17.2 Description

The method with a `seed` value starts by assigning the `seed` value to an internal accumulator. Then for each of the elements in the source stream, the next accumulator value is computed, by invoking the `binaryOperator` function, with the current accumulator value as the first argument and the current element as the second argument. The final accumulator value is returned.

The method without a `seed` value uses the first element of the source elements as the `seed` value. If the source stream is empty, an empty `Optional` is returned, otherwise an `Optional` with the final accumulator value is returned.

2.3.17.3 See

Section 2.3.3.2 “Optional” on page 2-48

Section 2.3.4.5 “binaryOperator” on page 2-49

2.3.17.4 Example

To find tallest student in a class:

```
students.stream().reduce((p,q)->(p.height>q.height? p: q).get()
```

2.3.18 max

2.3.18.1 Syntax

```
Optional<S> Stream<S>.max()
Optional<S> Stream<S>.max(((p,q)->int) comparator)
```

2.3.18.2 Description

This method computes the maximum of the elements in the source stream. If the `comparator` function is specified, it is used for comparisons. If no `comparator` function is specified, the elements themselves are compared, and must implement `Comparable`, otherwise an `ELException` is thrown.

This method returns an empty `Optional` for an empty stream.

2.3.18.3 See

Section 2.3.4.3 "comparator" on page 2-49

2.3.18.4 Examples

- To find tallest student in a class:

  ```
  students.stream().max((p,q)->p.height-q.height)
  ```

- To find the maximum height of the students in a class:

  ```
  students.stream().map(s->s.height).max()
  ```

2.3.19 min

2.3.19.1 Syntax

```
Optional<S> Stream<S>.min()
Optional<S> Stream<S>.min(((p,q)->int) comparator)
```

2.3.19.2 Description

This method computes the minimum of the elements in the source stream. If the `comparator` function is specified, it is used for comparisons. If no `comparator` function is specified, the elements themselves are compared, and must implement `Comparable`, otherwise an `ELException` is thrown.

This method returns an empty `Optional` for an empty stream.

2.3.19.3 See

Section 2.3.4.3 “comparator” on page 2-49

2.3.20 average

2.3.20.1 Syntax

```
Optional<S> Stream<S>.average()
```

2.3.20.2 Description

This method computes the average of all elements in the source stream by first computes the sum of the elements and then divides the sum by the number of elements. The elements are coerced to Number types according to Section 1.23.3 during the computation.

This method returns an empty `Optional` for an empty stream.

2.3.21 sum

2.3.21.1 Syntax

```
Number Stream<S>.sum()
```

2.3.21.2 Description

This method computes the sum of all elements in the source stream. The elements are coerced to Number types according to Section 1.23.3 during the computation.

This method returns zero for an empty stream.

2.3.22 count

2.3.22.1 Syntax

```
Long Stream<S>.count()
```

2.3.22.2 Description

This method returns the count of elements in the source stream.

2.3.23 anyMatch

2.3.23.1 Syntax

```
Optional<boolean> Stream<S>.anyMatch((S->boolean) predicate)
```

2.3.23.2 Description

This method returns an `Optional` of `true` if any element in the source stream satisfies the test given by the `predicate`. It returns an empty `Optional` if the stream is empty.

2.3.23.3 See

Section 2.3.4.1 “predicate” on page 2-49

2.3.23.4 Example

To determine if the list of integers contains any negative numbers:

```
integers.stream().anyMatch(i->i<0).orElse(false)
```

Note the use of `orElse` to set a default value for the empty list.

2.3.24 allMatch

2.3.24.1 Syntax

```
Optional<boolean> Stream<S>.allMatch((S->boolean) predicate)
```

2.3.24.2 Description

This method returns an `Optional` of `true` if all elements in the source stream satisfy the test given by the `predicate`. It returns an empty `Optional` if the stream is empty.

2.3.24.3 See

Section 2.3.4.1 “predicate” on page 2-49

2.3.25 noneMatch

2.3.25.1 Syntax

```
Optional<boolean> Stream<S>.noneMatch((S->boolean) predicate)
```

2.3.25.2 Description

This method returns an `Optional` of `true` if none of the elements in the source stream satisfies the test given by the `predicate`. It returns an empty `Optional` if the stream is empty.

2.3.25.3 See

Section 2.3.4.1 "predicate" on page 2-49

2.3.26 findFirst

2.3.26.1 Syntax

```
Optional<S> Stream<S>.findFirst()
```

2.3.26.2 Description

This method returns an `Optional` containing the first element in the stream, or an empty `Optional` if the stream is empty.

2.3.26.3 See

Section 2.3.3.2 "Optional" on page 2-48

CHAPTER **A**

Changes

This appendix lists the changes in the EL specification. This appendix is non-normative.

A.1 Changes between 3.0 Final Release and Proposed Final Draft

- Added Section 1.5.4, “Invoking method expressions”.
- Added Section 2.3.22, “count”.

A.2 Changes between 3.0 Proposed Final Draft and 3.0 EDR

- Complete rewrite of Chapter 2, to align with Java SE 8 libraries.
- Replace “+” and “cat” with “+=” as the String concatenation operator.
- Removed T(...) syntax as a type specifier.

A.3 New in 3.0 EDR

- Removed API from the specification document, since they are included in the javadocs.

- Added Chapter 2 “Operations on Collection Objects”.
- Added 1.8, String concatenation operator.
- Added 1.13, Assignment operator.
- Added 1.14, Semi-colon operator.
- Added 1.20 Lambda Expression.
- Added 1.22 Static Field and Methods.
- Added `T` and `cat` to 1.17 Reserved words.
- Modified 1.16 Operator precedence.
- Modified coercion rule from `nulls` to non-primitive types.
- Many changes to the javadoc API.

A.4 Incompatibilities between EL 3.0 and EL 2.2

EL 3.0 introduces many new features, and although we take care to keep it backward compatible, there are a few areas that cannot be made backward compatible, either because the new features requires it, or because the feature in EL 2.2 is a bug that needs to be fixed. An implementation can provide an option to revert to the 2.2 behavior, if desired.

- The default coercion for `nulls` to non-primitive types (except String) returns `nulls`. For instance, a `null` coerced to `Boolean` now returns a `null,` while a null coerced to boolean returns `false`.

A.5 Changes between Maintenance 1 and Maintenance Release 2

The main change in this release is the addition of method invocations with parameters in the EL, such as #{trader.buy(“JAVA”)}.

- Added one method in `javax.el.ELResolver`:
  - Object invoke(ELContext context, Object base, Object method, Class<?>[] paramTypes, Object[] params).
- Added one method in `javax.el.BeanELResolver`:

    - Object invoke(ELContext context, Object base, Object method, Class<?>[] paramTypes, Object[] params).
- Added one method in `javax.el.CompositeELResolver`:
    - Object invoke(ELContext context, Object base, Object method, Class<?>[] paramTypes, Object[] params).
- Section 1.1.1. Added to the first paragraph:

  Simlarly, . operator can also be used to invoke methods, when the method name is known, but the [] operator can be used to invoke methods dynamically
- Section 1.2.1. Change the last part of the last paragraph from

  Upon evaluation, the EL API verifies that the method conforms to the expected signature provided at parse time. There is therefore no coercion performed.

  to

  Upon evaluation, if the expected signature is provided at parse time, the EL API verifies that the method conforms to the expected signature, and there is therefore no coercion performed. If the expected signature is not provided at parse time, then at evaluation, the method is identified with the information of the parameters in the expression and the parameters are coerced to the respective formal types.
- Section 1.6

  Added syntax for method invocation with parameters.

  The steps for evaluation of the expression was modified to handle the method invocations with parameters.
- Section 1.19

  Production of `ValueSuffix` includes the optional parameters.

A.6 Changes between 1.0 Final Release and Maintenance Release 1

- Added two methods in `javax.el.ExpressionFactory:`
    - `newInstance()`
    - `newInstance(Properties)`

A.7 Changes between Final Release and Proposed Final Draft 2

Added support for enumerated data types. Coercions and comparisons were updated to include enumerated type types.

A.8 Changes between Public Review and Proposed Final Draft

New constructor for derived exception classes

Exception classes that extend `ELException` (`PropertyNotFoundException`, `PropertyNotWritableException`, `MethodNotFoundException`) did not have a constructor with both '`message`' and '`rootCause`' as arguments (as it exists in `ELException`). The constructor has been added to these classes.

`javax.el.ELContext` API changes

- removed the `ELContext` constructor
  `protected ELContext(javax.el.ELResolver resolver)`
- added the following abstract method in `ELContext`
  `public abstract javax.el.ELResolver getELResolver();`

Section 1.8.1 - A {<,>,<=,>=,lt,gt,le,ge} B

- If the first condition (A==B) is false, simply fall through to the next step (do not return false). See See issue 129 at jsp-spec-public.dev.java.net.

`javax.el.ResourceBundleELResolver`

- New `ELResolver` class added to support easy access to localized messages.

Generics

- Since JSP 2.1 requires J2SE 5.0, we've modified the APIs that can take advantage of generics. These include:
  `ExpressionFactory:createValueExpression()`,
  `ExpressionFactory:createMethodExpression()`,
  `ExpressionFactory:coerceToType()`, `ELResolver:getType()`,
  `ELResolver:getCommonPropertyType()`, `MethodInfo:MethodInfo()`,
  `MethodInfo.getReturnType()`, `MethodInfo:getParamTypes()`

A.9 Changes between Early Draft Release and Public Review

New concept: EL Variables

The EL now supports the concept of EL Variables to properly support code structures such as <c:forEach> where a nested action accesses a deferred expression that includes a reference to an iteration variable.

- Resulting API changes are:
  - `The javax.el` package description describes the motivation behind EL variables.
  - `ElContext` has two additional methods to provide access to `FunctionMapper` and `VariableMapper`.
  - `ExpressionFactory` creation methods now take an `ELContext` parameter. `FunctionMapper` has been removed as a parameter to these methods.
  - Added new class `VariableMapper`
- At a few locations in the spec, the term "variable" has been replaced with "model object" to avoid confusion between model objects and the newly introduced EL variables.
- Added new section “Variables” after section 1.15 to introduce the concept of EL Variables.

EL in a nutshell (section 1.1.1)

- Added a paragraph commenting on the flexibility of the EL, thanks to its pluggable API for the resolution of model objects, functions, and variables.

javax.el.ELException

- `ElException` now extends `RuntimeException` instead of `Exception`.
- Method `getRootCause()` has been removed in favor of `Throwable.getCause()`.

javax.el.ExpressionFactory

- Creation methods now use `ELContext` instead of `FunctionMapper` (see EL Variables above).
- Added method `coerceToType()`. See issue 132 at jsp-spec-public.dev.java.net.

javax.el.MethodExpression

- `invoke()` must unwrap an `InvocationTargetExceptions` before re-throwing as an `ELException`.

Section 1.6 - Operators [] and .

- `PropertyNotFoundException` is now thrown instead of `NullPointerException` when this is the last property being resolved and we're dealing with an lvalue that is null.

Section 1.13 - Operator Precedence

- Clarified the fact that qualified functions with a namespace prefix have precedence over the operators.

Faces Action Attribute and MethodExpression

In Faces, the `action` attribute accepts both a String literal or a `MethodExpression`. When migrating to JSF 1.2, if the attribute's type is set as `MethodExpression`, an error would be reported if a String literal is specified because a String literal cannot evaluate to a valid `javax.el.MethodExpression`.

To solve this issue, the specification of MethodExpression has been expanded to also support String literal-expressions. Changes have been made to:

- Section 1.2.2
- `ExpressionFactory.createMethodExpression()`
- `javax.el.MethodExpression:invoke()`