Expression Language Specification
Version 3.0 Public Review Release

Kin-man Chung, editor

Oracle Corporation
www.oracle.com

Public Review Release - June 18, 2012

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: Public Review

Release: June 18, 2012

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Oracle hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Oracle's intellectual property rights to:

1.Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.

2.Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:

(i) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;

(ii) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and

(iii) includes the following notice:

"This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Contents

Preface

This is the Expression Language specification version 3.0, developed 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 `Strings` (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.

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 the operator is a `+`, and either `A` or `B` is a String, then `+` is a string concatenation operator.
- 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 {+, cat} B`

The + operator is a string concatenation operator if and only if at least one of the operands is a String.

To evaluate `A + B or A cat 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 reolution, 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.

1.14 Semicolon Operator - `A ; B`

The semicolon operators 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)`
- `cat`
- `< > <= >= 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    T       cat
```

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`

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 referrenced 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 == 'Larry')`

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.

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

The syntax `T(className)`, where `className` is a full Java class name, denotes a Java class name at parse time. This by itself is not evaluated and does not produce a value. When followed by a "." and an identifier, the construct `T(className).id` denotes and evaluates to a static field or method (of the name `id`) of a class. For instance,

```
T(java.lang.Boolean).TRUE
```

evaluates to the value of the static field `java.lang.Boolean.TRUE`.

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

```
T(java.math.RoundingMode).FLOOR
```

1.22.1 Access Restrictions and Imports

For security, the following restrictions are enforced.

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

1.22.2 Imports of Classes and Packages

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 class that has been imported can be referenced without the package name, and without the T(...) syntax. The following syntaxes refer to the same static field.

```
T(java.lang.Boolean).TRUE
T(Boolean).TRUE
Boolean.TRUE
```

1.22.3 Special Fields and Methods

The field `class` refers to the `java.lang.Class` instance of the class. For instance, the expression

```
T(java.lang.Boolean).class
```

evaluates to the object `java.lang.Boolean.class`.

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

```
T(java.lang.Boolean)(true)
```

or

```
Boolean(true)
```

denotes the invocation of the constructor of the class with the supplied arguments.

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 also be specified in an `ELResolver` by implementing the method `convertToType`. More than one `ELResolvers` can be specified for performing conversion from object to different types, and they are 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, 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 `String`: return `A`
- Otherwise, if `A` is `null`: return `null`
- 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 '&', 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()
    | Type()
    | 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>
}
/*
 * Type
 * T(...)
 */
void Type() #Type : { Token t; StringBuilder buffer = new
StringBuilder(); }
```

```
{
        <TYPE> <LPAREN> t=<IDENTIFIER> {buffer.append(t.image); }
                 (<DOT> t=<IDENTIFIER>
{buffer.append('.').append(t.image);} )*
               <RPAREN> {jjtThis.setImage(buffer.toString()); }
        (MethodArguments())?
}
/*
 * 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 : "cat" >
|       < ASSIGN : "=" >
```

```
|       < TYPE : "T" >
|       < 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 the operations on collection objects in the Expression Language. It describes how collection objects and literals can be constructed in EL. It also describes the collection operators that are supported in EL.

2.1 Overview

To provide full support for collection objects, EL includes syntaxes for constructing sets, lists, and maps dynamically. These syntaxes resemble those in the Java Language (to be included in Java SE 8), and other scripting languages.

EL 3.0 provides an extensive list of operations on collections. These operations are implemented in the Expression Language with an ELResolver that intercepts and resolves calls to methods on collection objects to perform the necessary operations and return the desired results. These operations are really not part of the EL language, and can be taken as an example of what can be achieved with the EL language. The operations described in the specification served as a default implementation of the collection operations, and provide a standardized syntax for collection operations. However, an user can easily modify the names the behavior of the operations by providing hist/her own ELResolver.

Readers familiar with Language Integrated Query, LINQ, from Microsoft .NET framework will notice the similarities between the EL collection operations and LINQ operators. Some efforts was made to keep the operations the same, but there are some differences, see Section 2.3.1.

It is also anticipated that Java SE 8 will include some support for collection operations. EL will very likely evolve to include them.

Central to the implementation is the use of Lambda expressions, now supported in EL. Usually these Lambda expressions are applied to the elements of the collection object, when it is enumerated. For instance, in filter operators, a Lambda expression acts like a predicate function that determines if an item should be included in the resulting collection; or in projection operations, it acts like a selector function that selects the sub-field of the elements to be included in the resulting collection. These Lambda expressions are usually specified as arguments to the operators.

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

The concrete representation of a set is `java.lang.util.HashSet<Object>`.

2.2.1.1 Syntax

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

2.2.1.2 Example

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

2.2.2 List Construction

The concrete representation of a list is `java.lang.util.ArrayList<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

The concrete representation of a map is `java.lang.util.HashMap<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 Operators

The collection operators on collection objects are realized as methods calls to the collection in the EL syntax. To be precise, the ELResolver intercepts and resolves such a call if the base object implements `java.lang.Iterable`, and the property is an identifier whose name equals the name of an operator.

Since a Java array does not implement `java.lang.Iterable`, they are automatically converted to a `java.util.List` before calling the query operators.

A `java.util.Map` does not implement `java.lang.Iterable`. A collection view of a `Map`, such as `MapEntry` can be used as the base object for the operators. However, a `Map` is not automatically converted to an `Iterable`.

2.3.1 Differences Between EL and .NET syntaxes

Due to the differences in the Java and .NET languages, and also due to the different conventions used in the frameworks, there are some differences in the syntaxes.

The syntax for the Lambda expression in EL follows JDK 8, and is similar, and is not exactly the same as the one in C#. One notable difference is the use of -> instead of => to speparate the parameters from the body of a Lambda expression.

In most of the cases, a .NET interface can be roughly mapped to a Java type. For the cases where there is no mapping, they are defined in the package `javax.el`. The following is the mapping of all the types used in the operators.

TABLE 2-1 Mapping .NET Types

.NET Types	Java Types	javax.el Types
IEnumerable	java.lang.Iterable	
IDictionary	java.util.Map	
IGrouping		Grouping
Comparer	java.util.Comparator	
Lookup<K,V>	Map<Grouping<K,V>>	
InvalidOperationException		InvalidOperationException

2.3.2 Examples in this Chapter

To illustrate how the operators work, examples are provided. The examples are taken from the web page *http://msdn.microsoft.com/en-us/library/bb394939.aspx*, slightly simplified and modified to conform to the EL syntaxes.

The examples assume the following collections.

```
public class Product {
   int productID;
   String name;
   String category;
   double unitPrice;
   int unitsInStock;
}

public class Order {
   int orderID;
   int customerID;
   Date orderDate;
   double total;
}

public class Customer {
   int customerID;
   String name;
   String country;
   String phone;
   List<Order> orders;
}
```

The database contains the following records.

products:

productID	name	category	unitPrice	unitsInStock
200	Eagle	book	12.5	100
201	Coming Home	dvd	8.0	50
202	Greatest Hits	cd	6.5	200
203	History of Golf	book	11.0	30
204	Toy Story	dvd	10.0	1000
205	iSee	book	12.5	150

customers:

customerID	name	country	phone	orders (orderID)
100	John Doe	USA	650-734-2187	10, 11, 12
101	Mary Lane	USA	302-145-8765	13, 14
102	Charlie Yeh	China	08-7565-2323	15

orders:

orderID	customerID	date	total
10	100	2/18/2010	20.80
11	100	5/3/2011	34.50
12	100	8/2/2011	210.75
13	101	1/15/2011	50.23
14	101	1/3/2012	126.77
15	102	4/5/2011	101.20

2.3.3 Operator Syntax Description

Since the operators are not really Java methods, their APIs cannot be specified in Javadocs. Instead, the syntax and the semantics of the operators are described in this chapter. Pseudo method declarations are used for the operators. It includes

- The return type
- The type of the base object
- The method name
- The method parameters

A typical method declaration would looks like

```
returnT baseT.method(T1 arg1, T2 arg2)
```

Some operators 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. By examining the number of arguments and their types specified in the EL expression, it is possible for a method to determine if which parameters are specified. A `null` parameter is not allowed.

Some of the parameters are Lambda expressions, also known as functions. A Lambda expression can have parameters and returns a value. To describe the parameter types and the return type of a Lambda expression, the following notation is used.

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

For example, the declaration for the operator `where` is

```
Iterable<S> Iterable<S>.where((S->boolean) predicate)
```

From this we know that the source object is an `Iterable`, and the return object is also an `Iterable`, 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 collection, when iterated, and the function returns a `boolean`.

The generic types in the declaration is used only to help the reader of this document to identify the type relationships among the various parts of the declaration, and do not have the same meaning as those in the Java language. In reality, EL usually deal with collections of Objects, and does not track their generic types.

2.3.4 General Properties of the Operators

The operators operate on a source collection, and most of them also return a collection. A collection is represented in EL as a `java.lang.Iterable<Object>`. Therefore the result of one operator can be fed into another operator, and the operators can be chained together to achieve what is generally known as fluent syntax, as in the following example.

```
 products.where(p->p.unitPrice > 10).
          select(p->[p.name,p.unitPrice])
```

Because an operator returns an `Iterable` instead of a collection object, there is no need to construct the collection object in most of the cases. In the above example, it is not necessary for the `where` operator to copy the elements of `products` to another list. Instead, it only needs to yield the product elements that satisfy the condition and skip those that don't. Therefore the operators can implemented efficiently.

If an operator returns an `Iterable`, we say an element is yielded to mean that it is added as an element of the collection being returned.

Because the `Iterable` returned from an operator is executed only when it is enumerated, the execution of the operator is deferred. This can lead to surprises, if the nature of the deferred execution is not understood. For instance, if the result of the query in the above example is saved, and a new product is then added to `products`. If we enumerate the saved `Iterable`, the result will include the added new product, if it's unit price is greater than 10!

The conversion operators, such as toList or toMap, should be used to convert an Iterable to a collection object, and to disable the deferred execution.

Another property of the operators is that they do not change the input collection objects. Temporary collection objects are always constructed when necessary.

2.3.5 where Operator

The `where` operator filters the elements of a collection based on a predicate.

2.3.5.1 Syntax

```
Iterable<S> Iterable<S>.where((S->boolean) predicate)
Iterable<S> Iterable<S>.where((S,int)->boolean) predicate)
```

2.3.5.2 Description

The `where` operator iterates over the source elements and yields those elements for which the `predicate` function returns `true`. The first argument of `predicate` function represents the element to test. The second argument, if present, represents the zero-based index of the element within the source collection.

2.3.5.3 Example

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

```
products.where(p->p.unitPrice >= 10)
```

The result is

```
 Product: 200, Eagle, book, 12.5, 100
 Product: 203, History of Golf, book, 11.0, 30
 Product: 204, Toy Story, dvd, 10.0, 1000
 Product: 205, iSee, book, 12.5, 150
```

2.3.6 select Operator

The `select` operator performs a projection over a collection.

2.3.6.1 Syntax

```
Iterable<R> Iterable<S>.select((S->R) selector)
Iterable<R> Iterable<S>.select(((S,int)->R) selector)
```

2.3.6.2 Description

The `select` operator iterates over the source elements and yields the results of evaluating the `selector` function for each element. The first argument of `selector` represents the element to process. The second argument, if present, represents the zero-based index of the element within the source collection.

2.3.6.3 Example

To create a collection of the names of all products:

```
products.select(p->p.name)
```

The result is

```
 Eagle
 Coming Home
 Greatest Hits
 History of Golf
 Toy Story
 iSee
```

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

```
products.where(p->p.unitPrice >= 10).
         select(p->[p.name,p.unitPrice])
```

The result is

```
 [Eagle, 12.5]
 [History of Golf, 11.0]
 [Toy Story, 10.0]
 [iSee, 12.5]
```

2.3.7 selectMany Operator

The `selectMany` operator performs a one-to-many element projection over a collection.

2.3.7.1 Syntax

```
Iterable<R> Iterable<S>.selectMany((S->Iterable<R>) selector)
Iterable<R> Iterable<S>.selectMany(((S,int)->Iterable<R>) selector)
Iterable<R> Iterable<S>.selectMany((S->Iterable<I>) selector,
                                   ((S,I)->R) resultSelector)
Iterable<R> Iterable<S>.selectMany(((S,int)->Iterable<I>) selector,
                                   ((S,I)->R) resultSelector)
```

2.3.7.2 Description

The `selectMany` operator iterates over the source elements and for each element (called the outer element), the `selector` function is evaluated to produce an `Iterable` (called inner `Iterable`). The behavior would be undefined if the result of evaluating the `selector` function is not an `Iterable`. The inner `Iterable` is then iterated, and its element (called the inner element) is yielded, if there is no `resultSelector` function. If there is a `resultSelector` function, it is invoked, with the outer and the inner element as its arguments, and the result is yielded. The second argument to `selector` function, if present, represents the zero-based index of the element within the source collection.

Roughly speaking, `selectMany` flattens a `List<List<R>>` into a `List<R>`.

2.3.7.3 Example

To find all orders of the customers in USA:

```
customers.where(c->c.country == 'USA').
          selectMany(c->c.orders)
```

The result is

```
Order: 10, 100, 2/18/2010, 20.8
Order: 11, 100, 5/3/2011, 34.5
Order: 12, 100, 8/2/2011, 210.75
Order: 13, 101, 1/15/2011, 50.23
Order: 14, 101, 1/3/2012, 126.77
```

To create a list of customer names and order IDs of the orders placed in 2011 by customers in USA:

```
customers.where(c->c.country == 'USA').
          selectMany(c->c.orders, (c,o)->{'o':o,'c':c}).
          where(co->co.o.orderDate.year == 2011).
          select(co->[co.c.name, co.o.orderID])
```

The result is

```
[John Doe, 11]
[John Doe, 12]
[Mary Lane, 13]
```

Note the creation of an intermediate map so that we can retrieve the customer name given the customerID. Alternately, we can also do the following to get the same result. Note the use of nested Lambda expression in it.

```
customers.where(c->c.country == 'USA').
          selectMany(c->c.orders).
          where(o->o.orderDate.year == 2011).
          select(o->[customers.where(c->c.customerID==o.customerID).
                              select(c->c.name).
                              single(),
                     o.orderID])
```

2.3.8 take Operator

The `take` operator yields a given number of elements from a collection and then skips the rest.

2.3.8.1 Syntax

```
Iterable<R> Iterable<S>.take(int count)
```

2.3.8.2 Description

The `take` operator iterates over the source elements and yields first `count` number of elements. If `count` is greater than the number of source elements, all the elements are yielded. If the `count` is less than or equal to zero, no elements are yielded.

2.3.8.3 Example

To find the 3 most expensive products:

```
products.orderByDescending(p->p.unitPrice).
         take(3)
```

The result is

```
 Product: 200, Eagle, book, 12.5, 100
 Product: 205, iSee, book, 12.5, 150
 Product: 203, History of Golf, book, 11.0, 30
```

2.3.9 skip Operator

The `skip` operator skips a given number of elements from a collection and then yields the rest.

2.3.9.1 Syntax

```
Iterable<R> Iterable<S>.skip(int count)
```

2.3.9.2 Description

The `skip` operator iterates over the source elements, skipping the first `count` elements and yielding the rest. If the source collection contains fewer than `count` elements, nothing is yielded. If `count` is less than or equal to zero, all elements of the source collection are yielded.

2.3.10 takeWhile Operator

The `takeWhile` operator yields elements from a collection while a test is true and then skips the rest.

2.3.10.1 Syntax

```
Iterable<R> Iterable<S>.takeWhile((S->boolean) predicate)
Iterable<R> Iterable<S>.takeWhile((S,int)->boolean) predicate)
```

2.3.10.2 Description

The `takeWhile` operator iterates over the source elements, testing each element using the `predicate` function and yielding the element if the result is `true`. The iteration stops when the `predicate` function returns `false` or the end of the source elements is reached. The first argument of the `predicate` function represents the element to test. The second argument, if present, represents the zero-based index of the element within the source collection.

2.3.11 skipWhile Operator

The `skipWhile` operator skips elements from a sequence while a test is true and then yields the rest.

2.3.11.1 Syntax

```
Iterable<R> Iterable<S>.skipWhile((S->boolean) predicate)
Iterable<R> Iterable<S>.skipWhile((S,int)->boolean) predicate)
```

2.3.11.2 Description

The `skipWhile` operator iterates over the source elements, testing each element using the `predicate` function and skipping the element if the result is `true`. Once the `predicate` function returns `false` for an element, that element and the remaining elements are yielded with no further invocations of the `predicate` function. If the `predicate` function returns `true` for all elements in the collection, no elements are yielded. The first argument of the `predicate` function represents the element to test. The second argument, if present, represents the zero-based index of the element within the source sequence.

2.3.12 join Operator

The `join` operator performs an inner join of two collections based on matching keys extracted from the elements.

2.3.12.1 Syntax

```
Iterable<R> Iterable<S>.join(Iterable<I> innerSequence,
                             (S->K) outerKeySelector,
                             (I->K) innerKeySelector,
                             ((S,I)->R) resultSelector)
Iterable<R> Iterable<S>.join(Iterable<I> innerSequence,
                             (S->K)outerKeySelector,
                             (I->K) innerKeySelector,
                             ((S,I)->R} resultSelector,
                             Comparator comparator)
```

2.3.12.2 Description

The `join` operator iterates over the source elements, and for each element (called outer element), the `outerKeySelector` function is evaluated, to a value (called key1). For each non-null key1, `innerKeySequence` is iterated, and for each element (called inner element), the `innerKeySelector` function is evaluated, to a value (called key2). The value key1 is compared with key2, and if equal, the `resultKeySelector` function is evaluated (with outer element and inner element as its arguments), and the result is yielded.

If `comparator` is specified, it is used for the comparison of key1 to key2.

2.3.12.3 Example

To join customers and orders on their customer ID property, producing a list of customer names, order dates, and order totals:

```
customers.join(orders, c->c.customerID, o->o.customerID,
               (c,o)->[c.name, o.orderDate, o.total])
```

The result is

```
[John Doe, 2/18/2010, 20.8]
[John Doe, 5/3/2011, 34.5]
[John Doe, 8/2/2011, 210.75]
```

```
[Mary Lane, 1/15/2011, 50.23]
[Mary Lane, 1/3/2012, 126.77]
[Charlie Yeh, 4/15/2011, 101.2]
```

2.3.13 groupJoin Operator

The `groupJoin` operator performs a grouped join of two collections based on matching keys.

2.3.13.1 Syntax

```
Iterable<R> Iterable<S>.groupJoin(Iterable<I>innerSequence,
                                  (S->K)outerKeySelector,
                                  (I->K)innerKeySelector,
                                  ((S,Iterable<I>)->R)resultSelector)
Iterable<R> Iterable<S>.groupJoin(Iterable<I>innerSequence,
                                  (S->K)outerKeySelector,
                                  (I->K)innerKeySelector,
                                  ((S,Iterable<I>)->R}resultSelector,
                                  Comparator comparator)
```

2.3.13.2 Description

The `groupJoin` iterates over the source elements, and for each element (called outer element), the `outerKeySelector` function is evaluated, to a value (called key1). If key1 is non-null, `innerSequence` is iterated, and for each element (called inner element), the `innerKeySelector` function is evaluated, to a value (called key2). The inner elements whose key2 equals key1 are collected in a list. The `resultSelector` function is then evaluated, with the outer element and the list (can possibly be empty) as its arguments, and the result is yielded.

If `comparator` is specified, it is used for the comparison of key1 to key2.

The `groupJoin` operator preserves the order of the outer elements, and for each outer element, it preserves the order of the matching inner elements.

2.3.13.3 Example

To get a list customer names and total of his/her orders:

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

The result is

```
[John Doe, 266.05]
[Mary Lane, 177.0]
[Charlie Yeh, 101.2]
```

The same result can be obtained using `groupJoin` if the `orders` list is not kept with the customer record:

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

2.3.14 concat Operator

The concat operator concatenates two collections.

2.3.14.1 Syntax

```
Iterable<R> Iterable<S>.concat(Iterable<S> second)
```

2.3.14.2 Description

The `concat` operator iterates over the source elements, yielding each element, and then it iterates over the second collection, yielding each element.

2.3.15 orderBy, thenBy, orderByDescending and thenByDescending Operators

Operators in these operators order a collection according to one or more keys.

2.3.15.1 Syntax

```
Iterable<S> Iterable<S>.orderBy((S->K) keySelector)
Iterable<S> Iterable<S>.orderBy((S->K) keySelector,
                                Comparator comparator)
Iterable<S> Iterable<S>.orderByDescending((S->K) keySelector)
Iterable<S> Iterable<S>.orderByDescending((S->K) keySelector,
```

```
                                                   Comparator comparator)
Iterable<S> Iterable<S>.thenBy((S->K) keySelector)
Iterable<S> Iterable<S>.thenBy((S->K) keySelector,
                                      Comparator cmp)
Iterable<S> Iterable<S>.thenByDescending((S->K) keySelector)
Iterable<S> Iterable<S>.thenByDescending((S->K) keySelector,
                                                 Comparator comparator)
```

2.3.15.2 Description

The `orderBy`, `orderByDescending`, `thenBy`, and `thenByDescending` operators can be composed to order a collection by multiple keys. A composition of the operators has the form

```
source.orderBy(...).thenBy(...).thenBy(...) ...
```

where `orderBy(...)` is an invocation of `orderBy` or `orderByDescending` and each `thenBy(...)`, if any, is an invocation of `thenBy` or `thenByDescending`. The initial `orderBy` or `orderByDescending` establishes the primary ordering, the first `thenBy` or `thenByDescending` establishes the secondary ordering, the second `thenBy` or `thenByDescending` establishes the tertiary ordering, and so on. Each ordering is defined by:

- A `keySelector` function that extracts the key value from an element.
- An optional `comparator` for comparing key values. If no `comparator` is specified, the key value must implements `java.lang.Comparable`, otherwise no sorting will be done with this key values.
- A sort direction. The `orderBy` and `thenBy` methods establish an ascending ordering, the `orderByDescending` and `thenByDescending` methods establish a descending ordering.

Calling `orderBy` or `orderByDescending` on the result of an `orderBy/thenBy` operator will introduce a new primary ordering, disregarding the previously established ordering.

These operators perform a stable sort: if the key values of two elements are equal, the order of the elements is preserved.

The order of the original collection is not affected by these operators.

2.3.15.3 Examples

To list products, ordered first by category, then by descending price, and then by name.

```
products.orderBy(p->p.category).
         thenByDescending(p->p.unitPrice).
         thenBy(p->p.name)
```

The result is:

```
Product: 200, Eagle, book, 12.5, 100
Product: 205, iSee, book, 12.5, 150
Product: 203, History of Golf, book, 11.0, 30
Product: 202, Greatest Hits, cd, 6.5, 200
Product: 204, Toy Story, dvd, 10.0, 1000
Product: 201, Coming Home, dvd, 8.0, 50
```

To list products ordered by case insensitive name:

```
products.orderBy(p->p.name,
                 T(java.lang.String).CASE_INSENSITIVE_ORDER)
```

The result is:

```
Product: 201, Coming Home, dvd, 8.0, 50
Product: 200, Eagle, book, 12.5, 100
Product: 202, Greatest Hits, cd, 6.5, 200
Product: 203, History of Golf, book, 11.0, 30
Product: 205, iSee, book, 12.5, 150
Product: 204, Toy Story, dvd, 10.0, 1000
```

2.3.16 reverse Operator

The `reverse` operator reverses the elements of a collection.

2.3.16.1 Syntax

```
Iterable<S> Iterable<S>.reverse()
```

2.3.16.2 Description

The `reverse` operator iterates over the source elements, collecting all elements in a list, and then yields the elements of the list in reverse order.

2.3.17 groupBy Operator

The `groupBy` operator groups the elements of a collection.

2.3.17.1 Syntax

```
Iterable<Grouping<K,S>> Iterable<S>.groupBy((S->K) keySelector)
Iterable<Grouping<K,S>> Iterable<S>.groupBy((S->K) keySelector,
                                            Comparator comparator)
Iterable<Grouping<K,E>> Iterable<S>.groupBy((S->K) keySelector,
                                            (S->E) elementSelector)
Iterable<Grouping<K,E>> Iterable<S>.groupBy((S->K) keySelector,
                                            (S->E) elementSelector,
                                            Comparator comparator)
```

2.3.17.2 Description

The `groupBy` operator iterates over the source elements, and for each element, evaluates the `keySelector` and `elementSelector` (if present) functions. The `keySelector` and the `elementSelector` (if present) selects a key and a destination element, respectively, from the source elements. If no `elementSelector` is specified, the source elements become the destination elements. The destination elements with the same key value is grouped in a `Grouping`, which is then yielded.

The groupings are yielded in the order in which their key values first occurred in the source elements, and destination elements within a grouping are ordered by their occurrences in the source elements.

If `comparator` is specified, it is used for the comparison of key values.

2.3.17.3 Example

To group all product names by product category:

```
products.groupBy(p->p.category, p->p.name)
```

The result is:

```
 book: [Eagle, History of Golf, iSee]
 dvd: [Coming Home, Toy Story]
 cd: [Greatest Hits]
```

2.3.18 distinct Operator

The `distinct` operator eliminates duplicate elements from a collection.

2.3.18.1 Syntax

```
Iterable<S> Iterable<S>.distinc()
Iterable<S> Iterable<S>.distict(Comparator comparator)
```

2.3.18.2 Description

The `distinct` iterates over the source elements, yielding each element that has not previously been yielded.

If a `comparator` is specified, it is used to compare the elements.

2.3.18.3 Example

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

The result is

```
a
b
c
```

2.3.19 union Operator

The `union` operator produces the set union of two collections.

2.3.19.1 Syntax

```
Iterable<S> Iterable<S>.union(Iterable<S> second)
Iterable<S> Iterable<S>.union(Iterable<S> second,
                              Comparator comparator)
```

2.3.19.2 Description

The `union` iterates over the elements of the source and second collections, in that order, yielding each element that has not previously been yielded.

If a `comparator` is specified, it is used to compare the elements.

2.3.19.3 Example

```
['a', 'b', 'b', 'c'].union(['b', 'c', 'd'])
```

The result is

```
a
b
c
d
```

2.3.20 intersect Operator

The `intersect` operator produces the set intersection of two collections.

2.3.20.1 Syntax

```
Iterable<S> Iterable<S>.intersect(Iterable<S> second)
Iterable<S> Iterable<S>.intersect(Iterable<S> second,
                                  Comparator comparator)
```

2.3.20.2 Description

The `intersect` iterates over the source elements, yielding each element that has not previously been yielded, and which is also contained in the second collection.

If a `comparator` is specified, it is used to compare the elements.

2.3.20.3 Example

```
['a', 'b', 'b', 'c'].intersect(['b', 'c', 'd'])
```

The result is

```
b
c
```

2.3.21 except Operator

The `except` operator produces the set difference between two collections.

2.3.21.1 Syntax

```
Iterable<S> Iterable<S>.except(Iterable<S> second)
Iterable<S> Iterable<S>.except(Iterable<S> second,
                               Comparator comparator)
```

2.3.21.2 Description

The `except` operator iterates over the source elements, yielding each element that has not previously been yielded, and which is not contained in the second collection.

If a `comparator` is specified, it is used to compare the elements.

2.3.21.3 Example

```
['x', 'b', 'a', 'b', 'c'].except(['b', 'c', 'd'])
```

The result is

```
x
a
```

2.3.22 toArray Operator

The `toArray` operator creates an array from a collection.

2.3.22.1 Syntax

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

2.3.22.2 Description

The `toArray` operator iterates over the source elements and returns an array containing the elements.

2.3.23 toSet Operator

The `toSet` operator creates a `Set` from a collection.

2.3.23.1 Syntax

```
Set<S> Iterable<S>.toSet()
```

2.3.23.2 Description

The `toSet` operator iterates over the source elements and returns a `Set` containing the elements.

2.3.24 toList Operator

The `toList` operator creates a `List` from a collection.

2.3.24.1 Syntax

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

2.3.24.2 Description

The `toList` operator iterates over the source elements and returns a `List` containing the elements.

2.3.25 toMap Operator

The `toMap` operator creates a `Map` from a collection.

2.3.25.1 Syntax

```
Map<K, S> Iterable<S>.toMap((S->K) keySelector)
Map<K, V> Iterable<S>.toMap((S->K) keySelector,
                             (S->V) elementSelector)
```

2.3.25.2 Description

The `toMap` operator iterates over the source elements, and for each element, evaluates the `keySelector` and `elementSelector` functions to produce a key and a value. The resulting key and value pairs are returned in a `Map`. If no `elementSelector` was specified, the value is the element itself.

Since a `Map` cannot have duplicate keys, only the last unique key is mapped.

2.3.25.3 Example

```
orders.where(o->o.orderDate.year == 2011).
       toMap(o->o.orderID)
```

The result is

```
11=Order: 11, 100, 5/3/2011, 34.5
12=Order: 12, 100, 8/2/2011, 210.75
13=Order: 13, 101, 1/15/2011, 50.23
15=Order: 15, 102, 4/15/2011, 101.2
```

2.3.26 toLookup Operator

The `toLookup` operator creates a `Map<Grouping>` from a collection.

2.3.26.1 Syntax

```
Map<K,Grouping<K,S>> Iterable<S>.toLookup((S->K) keySelector)
Map<K,Grouping<K,V>> Iterable<S>.toLookup((S->K) keySelector,
                                          (S->V) elementSelector)
```

2.3.26.2 Description

The `toLookup` operator iterates over the source elements, and for each elements, evaluates the `keySelector` and `elementSelector` functions to produce a key and a value. The values with the same key are grouped in a `Grouping`, and the resulting key and the `Grouping` pairs are returned in a `Map`. If no `elementSelector` is specified, the value for each element is the element itself.

2.3.26.3 Example

```
products.toLookup(p->p.category, p->p.name)
```

The result is

```
book=book: [Eagle, History of Golf, iSee]
dvd=dvd: [Coming Home, Toy Story]
cd=cd: [Greatest Hits]
```

2.3.27 sequenceEqual Operator

The `sequenceEqual` operator checks whether two collections are equal.

2.3.27.1 Syntax

```
Iterable<S>.sequenceEqual(Iterable<S> second)
Iterable<S>.sequenceEqual(Iterable<S> second, Comparator comparator)
```

2.3.27.2 Description

The `sequenceEqual` operator iterates over the two source elements in parallel and compares corresponding elements. The method returns `true` if all corresponding elements compare equal and the two collections are of equal length. Otherwise, the method returns `false`.

If a `comparator` is specified, it is used to compare the elements.

2.3.28 first Operator

The `first` operator returns the first element of a collection.

2.3.28.1 Syntax

```
S Iterable<S>.first()
S Iterable<S>.first((S->boolean) predicate)
```

2.3.28.2 Description

The `first` operator iterates the source elements and returns the first element for which the `predicate` function returns `true`. If no `predicate` function is specified, the `first` operator simply returns the first element.

An `InvalidOperationException` is thrown if no element matches the predicate or if the source collection is empty.

2.3.29 firstOrDefault Operator

The `firstOrDefault` operator returns the first element of a collection, or a default value of `null` if no element is found.

2.3.29.1 Syntax

```
S Iterable<S>.firstOrDefault()
S Iterable<S>.firstOrDefault((S->boolean) predicate)
```

2.3.29.2 Description

The `firstOrDefault` operator iterates over the source elements and returns the first element for which the `predicate` function returns `true`. If no `predicate` function is specified, the `firstOrDefault` operator simply returns the first element of the sequence.

If no element matches the predicate or if the source sequence is empty, a `null` is returned.

2.3.30 last Operator

The `last` operator returns the last element of a collection.

2.3.30.1 Syntax

```
S Iterable<S>.last()
S Iterable<S>.last((S->boolean) predicate)
```

2.3.30.2 Description

The `last` operator iterates over the source elements, and returns the last element for which the `predicate` function returns `true`. If no `predicate` function is specified, the `last` operator simply returns the last element of the sequence.

An `InvalidOperationException` is thrown if no element matches the predicate or if the source collection is empty.

2.3.31 lastOrDefault Operator

The `lastOrDefault` operator returns the last element of a collection, or a default value of `null` if no element is found.

2.3.31.1 Syntax

```
S Iterable<S>.lastOrDefault()
```

```
S Iterable<S>.lastOrDefault((S->boolean) predicate)
```

2.3.31.2 Description

The `lastOrDefault` operator iterates over the source elements and returns the last element for which the `predicate` function returns `true`. If no `predicate` function is specified, the `lastOrDefault` operator simply returns the last element of the sequence.

If no element matches the predicate or if the source sequence is empty, a `null` is returned.

2.3.32 single Operator

The `single` operator returns the single element of a collection.

2.3.32.1 Syntax

```
S Iterable<S>.single()
```

```
S Iterable<S>.single((S->boolean) predicate)
```

2.3.32.2 Description

The `single` operator iterates over the source elements.

If a `predicate` function is specified, the `single` operator returns the single element for which the `predicate` function returns `true`. An `InvalidOperationException` is thrown if the source sequence contains no matching element or more than one matching element.

If no `predicate` function is specified, the `single` operator simply returns the single element of the sequence. An `InvalidOperationException` is thrown if the source sequence is empty or contions more than one element.

2.3.33 singleOrDefault Operator

The `singleOrDefault` operator returns the single element of a collection, or a default value of `null` if no element is found.

2.3.33.1 Syntax

```
S Iterable<S>.singleOrDefault()
S Iterable<S>.singleOrDefault((S->boolean) predicate)
```

2.3.33.2 Description

The `singleOrDefault` operator iterates over the source elements.

If a `predicate` function is specified, the `singleOrDefault` operator returns the single element for which the `predicate` function returns `true`. An `InvalidOperationException` is thrown if the source sequence contains more than one matching element. If no element matches the `predicate`, `null` is returned.

If no `predicate` function is specified, the `singleOrDefault` operator simply returns the single element of the sequence. An `InvalidOperationException` is thrown if the source sequence contains more than one element. If the source sequence is empty, `null` is returned.

2.3.34 elementAt Operator

The `elementAt` operator returns the element at a given index in a sequence.

2.3.34.1 Syntax

```
S Iterable<S>.elementAt(int index)
```

2.3.34.2 Description

The `elementAt` operator first checks whether the source collection implements `List`. If it does, `List.get()` is used to obtain the element at the given index. Otherwise, the source elements are iterated until `index` elements have been skipped, and the element found at that position is returned.

An `IndexOutOfBoundsException` is thrown if the index is less than zero or greater than or equal to the number of elements in the collection.

2.3.35 elementAtOrDefault Operator

The `elementAtOrDefault` operator returns the element at a given index in a collection, or a default value of `null` if the index is out of range.

2.3.35.1 Syntax

```
S Iterable<S>.elementAtOrDefault(int index)
```

2.3.35.2 Description

The `elementAtOrDefault` operator first checks whether the source collection implements `List`. If it does, `List.get()` is used to obtain the element at the given index. Otherwise, the source elements are iterated until `index` elements have been skipped, and the element found at that position is returned.

A `null` is returned if the index is less than zero or greater than or equal to the number of elements in the collection.

2.3.36 defaultIfEmpty Operator

The `defaultIfEmpty` operator supplies a default element for an empty collection.

2.3.36.1 Syntax

```
Iterable<S> Iterable<S>.defaultIfEmpty()
```

```
Iterable<S> Iterable<S>.defaultIfEmpty(S defaultValue)
```

2.3.36.2 Description

The `defaultIfEmpty` operator iterates over the source elements and yields its elements. If the source collection is empty, the `defaultValue` is yielded. If `defaultvalue` is not specified, a `null` is yielded.

2.3.37 any Operator

The `any` operator checks whether any element of a collection satisfies a condition.

2.3.37.1 Syntax

```
boolean Iterable<S>.any()
boolean Iterable<S>.any((S->boolean) predicate)
```

2.3.37.2 Description

The `any` operator iterates over the source elements and returns `true` if any element satisfies the test given by the `predicate`. If no `predicate` function is specified, the `any` operator simply returns `true` if the source collection contains any elements.

The iteration stops as soon as the result is known.

2.3.38 all Operator

The `all` operator checks whether all elements of a collection satisfies a condition.

2.3.38.1 Syntax

```
boolean Iterable<S>.all((S->boolean) predicate)
```

2.3.38.2 Description

The `all` operator iterates over the source elements and returns `true` if all elements satisfies the test given by the `predicate`.

The iteration stops as soon as the result is known.

2.3.39 contains Operator

The `contains` operator checks whether a collection contains a given element.

2.3.39.1 Syntax

```
boolean Iterable<S>.contains(S element)
```

2.3.39.2 Description

If the source collection implements `Collection`, the method `Collection.contains()` is invoked to obtain the result. Otherwise, the source elements are iterated to determine if it contains an element with the given value. The iteration stops as soon as a matching element is found.

2.3.40 count Operator

The `count` operator counts the number of elements in a collection.

2.3.40.1 Syntax

```
Number Iterable<S>.count()
Number Iterable<S>.count((S->boolean) predicate)
```

2.3.40.2 Description

If a `predicate` is not specified, and the source collection implements `Collection`, the method `Collection.size()` is used to obtain the element count. Otherwise, the source collection is iterated to count the number of elements.

If a `predicate` is specified, the `count` operator iterates over the source elements and counts the number of elements for which the `predicate` function returns `true`.

2.3.41 sum Operator

The `sum` operator computes the sum of a collection.

2.3.41.1 Syntax

```
Number Iterable<Number>.sum()
Number Iterable<S>.sum((S->Number) selector)
```

2.3.41.2 Description

The `sum` operator iterates over the source elements, invokes the `selector` function for each element, and computes the sum of the resulting values. If no `selector` function is specified, the sum of the elements themselves is computed.

The `sum` operator returns zero for an empty sequence. Furthermore, the operator does not include `null` values in the result.

2.3.42 min Operator

The `sum` operator computes the minimum value of a collection.

2.3.42.1 Syntax

```
Number Iterable<Number>.min()
Number Iterable<S>.min((S->Number) selector)
```

2.3.42.2 Description

The `min` operator iterates over the source elements, invokes the `selector` function for each element, and finds the minimum of the resulting values. If no `selector` function is specified, the minimum of the elements themselves is computed. The values to be compared must implement `Comparable`.

The `min` operator returns `null` for an empty collection.

2.3.43 max Operator

The `sum` operator computes the maximum value of a collection.

2.3.43.1 Syntax

```
Number Iterable<Number>.max()
Number Iterable<S>.max((S->Number) selector)
```

2.3.43.2 Description

The `max` operator iterates over the source elements, invokes the `selector` function for each element, and finds the maximum of the resulting values. If no `selector` function is specified, the maximum of the elements themselves is computed. The values to be compared must implement `Comparable`.

The `max` operator returns `null` for an empty collection.

2.3.44 average Operator

The `average` operator computes the average value of a collection.

2.3.44.1 Syntax

```
Number Iterable<Number>.average()
Number Iterable<S>.average((S->Number) selector)
```

2.3.44.2 Description

The `average` operator iterates over the source elements, invokes the `selector` function for each element, and computes the average of the resulting values. If no `selector` function is specified, the average of the elements themselves is computed.

The `average` operator returns `null` for an empty collection.

2.3.45 aggregate Operator

The `aggregate` operator applies a function over a collection.

2.3.45.1 Syntax

```
S Iterable<S>.aggregate(((S,S)->S)func)
```

```
A Iterable<S>.aggregate(A seed,((A,S)->A)func)
R Iterable<S>.aggregate(A seed,((A,S)->A)func,(A->R)resultSelector)
```

2.3.45.2 Description

The `aggregate` operators with a `seed` value start by assigning the `seed` value to an internal accumulator. They then iterate over the source elements, repeatedly computing the next accumulator value by invoking the specified function `func` with the current accumulator value as the first argument and the current sequence element as the second argument. At the end of the iteration, if a `resultSelector` is not specified, the final accumulator value is returned, otherwise `resultSelector` is invoked, with the final accumulator as its argument, and the result is returned.

The `aggregate` operator without a `seed` value uses the first element of the source elements as the `seed` value. If the source collection is empty, the `aggregate` operator without a `seed` value throws an `InvalidOperationException`.

2.3.46 forEach Operator

The `forEach` operator applies an action over a collection.

2.3.46.1 Syntax

```
Object Iterable<S>.forEach(((S)->void)action)

Object Iterable<S>.forEach(((S,int)->void)action)
```

2.3.46.2 Description

The `forEach` operators iterate over the source elements, invokes the `action` function for each element. The first argument of `action` represents the element to process. The second argument, if present, represents the zero-based index of the element within the source collection.

The `forEach` operators always return `null`.

2.3.46.3 Example

To output the product names and prices with indices.

```
products.forEach((p,idx)->System.out.println(
                    idx + ": " + p.name + ", " + p.unitPrice))
```

The output is

```
0: Eagle, 12.5
1: Coming Home, 8.0
2: Greatest Hits, 6.5
3: History of Golf, 11.0
4: Toy Story, 10.0
5: iSee, 12.5
```

2.3.47 collections:range Function

The `range` function generates a consecutive sequence of integers.

2.3.47.1 Syntax

```
Iterable<Integer> collections:range(int start, int count)
```

2.3.47.2 Description

When the object returned by `collections:range` function is iterated, it yields the integers `start, start+1, ..., start+count-1`.

An `IllegalArgumentException` is thrown if `count` is less than zero, or if `start+count-1` is greater than `Integer.MAX_VALUE`.

2.3.47.3 Example

To generate a List of the squares of the integers from 0 to 4.

```
collections:range(0,5).select(x->x*x).toList()
```

The result is

```
[0, 1, 4, 9, 16]
```

2.3.48 collections:repeat Function

The `repeat` function generates a sequence of multiple copies of an object.

2.3.48.1 Syntax

```
Iterable<S> collections:repeat(S element, int count)
```

2.3.48.2 Description

When the object returned by `collections:range` function is iterated, it yields the object `element count` number of times.

An `IllegalArgumentException` is thrown if `count` is less than zero.

CHAPTER A

Changes

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

A.1 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.2 Imcompatibilities 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.

- New key words: `T` and `cat`.
- The operator + is now overloaded and behaves like that in the Java language
- The default coercion for `nulls` to non-primitive types returns `nulls`. For instance, a `null` coerced to `String` now returns a `null`, instead of an empty `String`.

A.3 Changes between Maintenance 1 and Maintenance Release 2

The main change in this release is the addition of method invokations 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.4 Changes between 1.0 Final Release and Maintenance Release 1

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

A.5 Changes between Final Release and Proposed Final Draft 2

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

A.6 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.7 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()`