Expression Language Specification
Version 3.0 Early Draft Release

Kin-man Chung, editor

Oracle Corporation
www.oracle.com

Early Draft Release - March 19, 2012

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

Oracle Corporation
www.oracle.com

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

ORACLE IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY THEM, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

Specification: JSR-341 Expression Language

Version: 3.0

Status: Early Draft Review

Release: March 19, 2012

NOTICE

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

draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.

Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Oracle intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Oracle if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.oracle" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Oracle, the Oracle logo, Java are trademarks or registered trademarks of Oracle USA, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY ORACLE. ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF

MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE

SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. ORACLE MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ORACLE OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF ORACLE AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Oracle (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Oracle with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Oracle a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

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. The specification is the first JSR that defines the Expression Language as an independent specification, with no dependencies on other technologies.

Related Documentation

Implementors of the Expression Language and web developers may find the following documents worth consulting for additional information:.

JavaServer Pages (JSP)	http://java.sun.com/products/jsp
JSP Standard Tag Library (JSTL)	http://java.sun.com/products/jsp/jstl
JavaServer Faces (JSF)	http://java.sun.com/j2ee/javaserverfaces
Java Servlet Technology	http://java.sun.com/servlet
Java 2 Platform, Standard Edition	http://java.sun.com/j2se
Java 2 Platform, Enterprise Edition	http://java.sun.com/j2ee
JavaBeans	http://java.sun.com/beans

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 LINQ operators 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`,

- If `expr-a` is an identifier and either it is an EL variable, or it is not resolved by the `ELResolver`,
  - The EL variable is set with the `ValueExpression` for expr-b, without evaluation
  - Evaluate `expr-b` and return its value.
- Else evaluate `expr-b` and assign its value to `expr-a`, by invoking `ValueExpression.setValue()` for `expr-a`. Return the value for `expr-b`.

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`
- `?` `:`
- `=`
- `;`

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())}`

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 expressions 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 EL function. It can be evaluated immediately,

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

It can be assigned to a variable and be evaluated indirectly,

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

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

- `employees.where(e->e.firstName == 'Larry')`

When evaluating a Lambda expression, the values of the actual parameters are used in the evaluation of the expression in the Lambda body. The number of actual parameters must be equal to or more than the number the formal parameters. The extra parameters supplied are ignored.

A Lambda expression can be nested within another Lambda expression. for instance,

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

The scope of a Lambda parameter is the body of the Lambda expression. A Lambda parameter takes precedence over other EL variables, identifiers or Lambda parameters of the nesting Lambda expressions.

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, 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 EL environment.

```
java.lang.*
```

A class that has been imported can be referenced without the package name, without the T(...) bracket. 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`.

To invoke the constructor of a class, the syntax T(classname)() can be used, such as

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

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.

1.23.1 To Coerce a Value X to Type Y

- 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 `""`
- 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` 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

- 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

- 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

Note: This section will be updated in the public review draft.

The valid syntax for an expression depends on its type.

For value expressions, the parser first attempts to parse the expression using the `LValue` production. If parsing fails, the `ValueExpression` will be read-only and parsing is attempted again using the `RValue` production. For method expressions, the parser must use only the `MethodExpression` production.]

These productions take into consideration literal-expressions and composite expressions wherever they are accepted.

```
LValue            ::=    '${' LValueInner '}'
                     |   '#{' LValueInner '}'
LValueInner       ::=    Identifier
                     |   NonLiteralValuePrefix (ValueSuffix)*
RValue            ::=    (RValueComponent1)+
                     |   (RValueComponent2)+
RValueComponent1  ::=    '${' Expression '}'
                     |   LiteralExpression
RValueComponent2  ::=    '#{' Expression '}'
                     |   LiteralExpression
MethodExpression  ::=    LValue
LiteralExpression::=     (LiteralComponent)* ([$#])?
                         i.e., a string of any characters that
                         doesn't include ${ or #{ unless escaped by
                         \${ or \#{.
LiteralComponent  ::=    ([^$#\])*\([$#])?
                     |   ([^$#])*([$#][^{])
                     |   ([^$#])*
Expression        ::= Expression1 ExpressionRest?
ExpressionRest    ::= '?' Expression ':' Expression
Expression1       ::=    Expression BinaryOp Expression
                     |   UnaryExpression
BinaryOp          ::=    'and'
                     |   '&&'
                     |   'or'
                     |   '||'
                     |   '+'
                     |   '-'
                     |   '*'
                     |   '/'
                     |   'div'
                     |   '%'
                     |   'mod'
                     |   '>'
                     |   'gt'
                     |   '<'
```

```
                    |   'lt'
                    |   '>='
                    |   'ge'
                    |   '<='
                    |   'le'
                    |   '=='
                    |   ‘eq’
                    |   ‘!=’
                    |   ‘ne’
UnaryExpression  ::=    UnaryOp UnaryExpression
                    |    Value
UnaryOp          ::=    '-'
                    |   ‘!’
                    |   ‘not’
                    |   ‘empty’
Value            ::=    ValuePrefix (ValueSuffix)*
ValuePrefix      ::=    Literal
                    |   NonLiteralValuePrefix
NonLiteralValuePrefix ::= '(' Expression ')'
                    |   Identifier
                    |   FunctionInvocation
ValueSuffix      ::=    ‘.’ Identifier MethodParameters?
                    |   ‘[‘ Expression ‘]’ MethodParameters?
MethodParameters ::= '(' (Expression (‘,’ Expression )* )? ')'
Identifier       ::= Java language identifier
FunctionInvocation::=(Identifier ‘:’)? Identifier ‘(‘
                    ( Expression ( ‘,’ Expression )* )? ‘)’
Literal          ::=    BooleanLiteral
                    |   IntegerLiteral
                    |   FloatingPointLiteral
                    |   StringLiteral
                    |   NullLiteral
BooleanLiteral   ::=    'true'
                    |   ‘false’
StringLiteral    ::=    '([^'\]|\'|\\)*'
                    |   "([^”\]|\”|\\)*"
                    i.e., a string of any characters enclosed by
                    single or double quotes, where \ is used to
                    escape ', ",and \. It is possible to use single
                    quotes within double quotes, and vice versa,
                    without escaping.
IntegerLiteral   ::= [‘0’-’9’]+
```

```
FloatingPointLiteral::=  ([‘0’-’9’])+ ‘.’ ([‘0’-’9’])* Exponent?
                       |   ‘.’ ([‘0’-’9’])+ Exponent?
                       |   ([‘0’-’9’])+ Exponent?
Exponent          ::= [‘e’,’E’] ([‘+’,’-’])? ([‘0’-’9’])+
NullLiteral       ::= 'null'
```

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 LINQ operators that are supported in the 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 (proposed), and other scripting languages.

Language Integrated Query, LINQ, is a Microsoft .NET framework component that adds data querying capabilities to .NET languages. It defines a set of methods, called standard query operators that can used to perform operations, like projections or filtering, on collection objects.

EL 3.0 provides full support for the LINQ standard query operators. These operators are implemented in the EL by the use of an ELResolver that resolves calls to these methods on collection objects to perform the necessary operations and return the desired results. Therefore, these operators can be easily modified or extended.

Central to the implementation is the use of Lambda expressions, now supported in the 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 element 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 LINQ Standard Query Operators

The LINQ standard query operators on collection objects are implemented as methods calls to the collection. To be precise, the ELResolver only 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 operators.

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 the base object for the operators. However, a `Map` is not automatically converted to an `Iterable`.

A disclaimer: The examples and the operator descriptions are taken from the web page *http://msdn.microsoft.com/en-us/library/bb394939.aspx*, slightly modified to conform to the EL syntaxes, and for clarifications.

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 minor differences in the syntaxes.

All operators (methods) in EL start with a lower-case letter, in conformance with the Java language convention.

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 with the descriptions. The examples assume a collections of objects of the following classes.

```
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, as if they are overloaded. A `null` value for a parameter indicates the absence of the parameter.

Some of the parameters are Lambda expressions, also known as functions. To describe the the types and the names of the parameters, and the type of the result of a Lambda expression, we use following notation

```
(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 base 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 a argument. The argument of the Lambda expression is an element of the base object, when enumerated, and the result of the Lambda expression evaluation is a `boolean`.

The generic types in the declaration is used as a helper for identifying the type relations among the various parts of the declaration only. In reality, EL always use the most general type, `Object`, as the element type.

2.3.4 General Properties of the Operators

The input to most of the operators is a sequence of objects. Most operators also return a sequence. A sequence of object 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 referred to 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 need to yield the product elements that satisfy the condition and skip those that don't. Therefore the operators can implemented efficiently.

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 a sequence based on a predicate.re Operator.

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

When the `Iterable` returned by `where` is enumerated, it enumerates the source sequence and yields those elements for which the `predicate` function returns true. 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.

2.3.5.3 Example

To creates a sequence of those products that have a price 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 sequence.

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

When the `Iterable` returned by `select` is enumerated, it enumerates the source sequence and yields the results of evaluating the `selector` function for each element. The first argument of the `selector` represents the element to process. The second argument, if present, represents the zero-based index of the element within the source sequence.

2.3.6.3 Example

To creates a sequence 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 objects containing the name and price of each product 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 sequence.

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

When the `Iterable` returned by `selectMany` is enumerated, it enumerates the source sequence. For each element (called the outer element), the `selector` function is evaluated, and should results in 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 enumerated, and its element (called the inner element), is yielded, if there is no `resultSelector` function. If there is a `resultSelector` function, the outer and the inner element are passed as arguments to `resultSelector`, and the result of evaluating `resultSelector` is yielded. The second argument to the `selector` function, if present, represents the zero-based index of the element within the source sequence.

Roughly speaking, `selectMany` can be used to flatten a List<List<R>> into a List<R>.

2.3.7.3 Example

To create a sequence of the 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 sequence of objects containing the customer name and order ID 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 sequence and then skips the remainder of the sequence.

2.3.8.1 Syntax

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

2.3.8.2 Description

When the `Iterable` returned by `take` is enumerated, it enumerates the source sequence and yields elements until the number of elements given by the `count` argument have been yielded or the end of the source is reached. If the `count` argument is less than or equal to zero, the source sequence is not enumerated and no elements are yielded.

2.3.8.3 Example

To create a sequence of the most expensive 3 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 sequence and then yields the remainder of the sequence.

2.3.9.1 Syntax

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

2.3.9.2 Description

When the `Iterable` returned by `skip` is enumerated, it enumerate the source sequence, skipping the number of elements given by the `count` argument and yielding the rest. If the source sequence contains fewer elements than the number given by the `count` argument, nothing is yielded. If the `count` argument is less than or equal to zero, all elements of the source sequence are yielded.

2.3.10 takeWhile Operator

The `takeWhile` operator yields elements from a sequence while a test is true and then skips the remainder of the sequence.

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

When the `Iterable` returned by `takeWhile` is enumerated, it enumerates the source sequence, testing each element using the `predicate` function and yielding the element if the result was true. The enumeration stops when the `predicate` function returns false or the end of the source sequence 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 sequence.

2.3.11 skipWhile Operator

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

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

When the `Iterable` returned by `skipWhile` is enumerated, it enumerates the source sequence, testing each element using the `predicate` function and skipping the element if the result was 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 sequence, 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 sequences 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

When the `Iterable` returned by `join` is enumerated, it enumerates the source sequence, and for each element (called outer element), the `outerKeySelector` function is evaluated, to a value (called key1). For each non-zero key1, `innerKeySequence` is enumerated, 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 the arguments), and the resulting object is yielded.

If the Comparator `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 sequence of tuples with customer name, order date, and order total:

```
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 sequences based on matching keys extracted from the elements.

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

When the `Iterable` returned by `groupJoin` is enumerated, it enumerates the source sequence, and for each element (called outer element), the `outerKeySelector` function is evaluated, to a value (called key1). If key1 is non-zero, `innerSequence` is enumerated, 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 sequence. The outer element and this sequence (can possibly be empty) are passed to the `resultSelector` function, and the resulting object is yielded.

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

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

2.3.13.3 Example

To get a sequence of tuples with customer name and total of all 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 sequences.

2.3.14.1 Syntax

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

2.3.14.2 Description

The Iterable returned from "concat" enumerates the source sequence, yielding each element, and then it enumerates the second sequence, yielding each element.

2.3.15 orderBy, thenBy, orderByDescending and thenByDescending Operators

Operators in the orderBy/thenBy family of operators order a sequence 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 make up a family of operators that can be composed to order a sequence 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 or if the `comparator` argument is null, 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 object is not affected by these operators.

2.3.15.3 Examples

To create a sequence of all 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 create a sequence of all 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 sequence.

2.3.16.1 Syntax

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

2.3.16.2 Description

When the `Iterable` returned from `reverse` is enumerated, it enumerates the source sequence, collecting all elements, and then yields the elements of the source sequence in reverse order.

2.3.17 groupBy Operator

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

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

When the `Iterable` returned from `groupBy` is enumerated, it enumerates the source sequence, and evaluates the `keySelector` and `elementSelector` (if present) functions once for each source element. The `keySelector` argument specifies a function that extracts the key value from a source element. The `elementSelector` argument, if present, specifies a function that maps a source element to a destination element. If no `elementSelector` is specified, the source elements become the destination elements. Once all key and destination element pairs have been collected, the destination elements with the same key value is grouped in a `Grouping` object, which is then yielded.

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

Key values are compared using the `Comparator comparator`, if specified.

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 sequence.

2.3.18.1 Syntax

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

2.3.18.2 Description

When the `Iterable` returned from `distinct` is enumerated, it enumerates the source sequence, yielding each element that has not previously been yielded. If a non-null `comparer` argument is supplied, 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 sequences.

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

When the `Iterable` returned from `union` enumerates the source and second sequences, in that order, yielding each element that has not previously been yielded. If a non-null `comparator` argument is supplied, 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 sequences.

2.3.20.1 Description

When the `Iterable` returned from `intersect` enumerates the source sequence, yielding each element that has not previously been yielded, and which is also contained in the second sequence. If a non-null `comparator` argument is supplied, it is used to compare the elements.

2.3.20.2 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 sequences.

2.3.21.1 Syntax

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

2.3.21.2 Description

When the `Iterable` returned from `except` is enumerated, it enumerates the source sequence, yielding each element that has not previously been yielded, and which is not contained in the second sequence. If a non-null `comparator` argument is supplied, 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 sequence.

2.3.22.1 Syntax

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

2.3.22.2 Description

The `toArray` operator enumerates the source sequence and returns an array containing the elements of the sequence.

2.3.23 toList Operator

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

2.3.23.1 Syntax

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

2.3.23.2 Description

The `toList` operator enumerates the source sequence and returns a List containing the elements of the sequence.

2.3.24 toMap Operator

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

2.3.24.1 Syntax

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

2.3.24.2 Description

The `toMap` operator enumerates the source sequence and evaluates the `keySelector` and `elementSelector` functions for each element to produce that element's key and value. The resulting key and value pairs are returned in a `Map`. If no `elementSelector` was specified, the value for each element is simply the element itself.

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

2.3.24.3 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
```

2.3.25 toLookup Operator

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

2.3.25.1 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)
```

2.3.25.2 Description

The `toLookup` operator enumerates the source sequence and evaluates the `keySelector` and `elementSelector` functions for each element to produce that element's key and 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` was specified, the value for each element is simply the element itself.

2.3.25.3 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]
```

2.3.26 sequenceEqual Operator

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

2.3.26.1 Syntax

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

2.3.26.2 Description

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

If a non-null `comparator` argument is supplied, it is used to compare the elements.

2.3.27 first Operator

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

2.3.27.1 Syntax

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

2.3.27.2 Description

The `first` operator enumerates the source sequence 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 of the sequence.

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

2.3.28 firstOrDefault Operator

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

2.3.28.1 Syntax

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

2.3.28.2 Description

The `firstOrDefault` operator enumerates the source sequence 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.

2.3.29 last Operator

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

2.3.29.1 Syntax

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

2.3.29.2 Description

The `last` operator enumerates the source sequence 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 sequence is empty.

2.3.30 lastOrDefault Operator

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

2.3.30.1 Syntax

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

2.3.30.2 Description

The `lastOrDefault` operator enumerates the source sequence 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.

2.3.31 single Operator

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

2.3.31.1 Syntax

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

2.3.31.2 Description

The `single` operator enumerates the source sequence and returns the single element for which the `predicate` function returns true. 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, contains no matching element or more than one matching element.

2.3.32 singleOrDefault Operator

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

2.3.32.1 Syntax

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

2.3.32.2 Description

The `singleOrDefault` operator enumerates the source sequence and returns the single element for which the `predicate` function returns true. 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 matching element. If no element matches the predicate or if the source sequence is empty, `null` is returned.

2.3.33 elementAt Operator

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

2.3.33.1 Syntax

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

2.3.33.2 Description

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

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

2.3.34 elementAtOrDefault Operator

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

2.3.34.1 Syntax

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

2.3.34.2 Description

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

2.3.35 defaultIfEmpty Operator

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

2.3.35.1 Syntax

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

2.3.35.2 Description

When the `Iterable` returned by `defaultIfEmpty` is enumerated, it enumerates the source sequence and yields its elements. If the source sequence is empty, a single element with the given `defaultValue` is yielded. If `defaultvalue` is not specified, a single element of `null` is yielded.

2.3.36 any Operator

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

2.3.36.1 Syntax

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

2.3.36.2 Description

The `any` operator enumerates the source sequence 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 sequence contains any elements.

The enumeration of the source sequence is terminated as soon as the result is known.

2.3.37 all Operator

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

2.3.37.1 Syntax

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

2.3.37.2 Description

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

The enumeration of the source sequence is terminated as soon as the result is known.

2.3.38 contains Operator

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

2.3.38.1 Syntax

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

2.3.38.2 Description

If the source `Iterable` is a `Collection`, `Collection.contains()` is invoked to obtain the result. Otherwise, the source sequence is enumerated to determine if it contains an element with the given value. If a matching element is found, the enumeration of the source sequence is terminated at that point.

2.3.39 count Operator

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

2.3.39.1 Syntax

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

2.3.39.2 Description

The `count` operator without a `predicate` first checks whether the source sequence implements `Collection`. If it does, `Collection.size()` is used to obtain the element count. Otherwise, the source sequence is enumerated to count the number of elements.

The `count` operator with a `predicate` enumerates the source sequence and counts the number of elements for which the `predicate` function returns true.

2.3.40 sum Operator

The `sum` operator computes the sum of a sequence of numeric values.

2.3.40.1 Syntax

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

2.3.40.2 Description

The `sum` operator enumerates the source sequence, 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.

2.3.41 min Operator

The `sum` operator computes the sum of a sequence of numeric values.

2.3.41.1 Syntax

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

2.3.41.2 Description

The `min` operator enumerates the source sequence, 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 sequence.

2.3.42 max Operator

The `sum` operator computes the sum of a sequence of numeric values.

2.3.42.1 Syntax

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

2.3.42.2 Description

The `max` operator enumerates the source sequence, 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 sequence.

2.3.43 average Operator

The `average` operator computes the average of a sequence of numeric values.

2.3.43.1 Syntax

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

2.3.43.2 Description

The `average` operator enumerates the source sequence, 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 sequence.

2.3.44 aggregate Operator

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

2.3.44.1 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)
```

2.3.44.2 Description

The `aggregate` operators with a `seed` value start by assigning the `seed` value to an internal accumulator. They then enumerate the source sequence, 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. The operator without a `resultSelector` returns the final accumulator value as the result. The operator with a `resultSelector` passes the final accumulator value to the supplied `resultSelector` and returns the resulting value.

The `aggregate` operator without a `seed` value uses the first element of the source sequence as the `seed` value, but otherwise functions as described above. If the source sequence is empty, the `aggregate` operator without a `seed` value throws an `InvalidOperationException`.

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 Chaper 2, Operatrions 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.
- Many changes to the API.

A.2 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.3 Changes between 1.0 Final Release and Maintenance Release 1

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

A.4 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.5 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.6 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()`