Java Unified Expression Language

Table of contents

1. Welcome to JUEL!

JUEL is an implementation of the Unified Expression Language (EL), specified as part of the JSP 2.1 standard (JSR-245), which has been introduced in JEE5. Additionally, *JUEL* 2.2 implements the JSP 2.2 maintenance release specification for full JEE6 compliance.

Motivation

Once, the EL started as part of JSTL. Then, the EL made its way into the JSP 2.0 standard. Now, though part of JSP 2.1, the EL API has been separated into package `javax.el` and all dependencies to the core JSP classes have been removed.

In other words: the EL is ready for use in non-JSP applications!

Features

JUEL provides a lightweight and efficient implementation of the Unified Expression Language.

- High Performance – Parsing expressions is certainly the expected performance bottleneck. *JUEL* uses a hand-coded parser which is up to 10 times faster than the previously used (javacc) generated parser! Once built, expression trees are evaluated at highest speed.
- Pluggable Cache – Even if *JUEL*s parser is fast, parsing expressions is relative expensive. Therefore, it's best to parse an expression string only once. *JUEL* provides a default caching mechanism which should be sufficient in most cases. However, *JUEL* allows to plug in your own cache easily.
- Small Footprint – *JUEL* has been carefully designed to minimize memory usage as well as code size.
- Method Invocations – *JUEL* supports method invocations as in `${foo.matches('[0-9]+')}`. Methods are resolved and invoked using the EL's resolver mechanism. As of *JUEL* 2.2, method invocations are enabled by default.
- VarArg Calls – *JUEL* supports Java 5 VarArgs in function and method invocations. E.g., binding `String.format(String, String...)` to function `format` allows for `${format('Hey %s','Joe')}`. As of *JUEL* 2.2, VarArgs are enabled by default.
- Pluggable – *JUEL* can be configured to be transparently detected as EL implementation by a Java runtime environment or JEE application server. Using *JUEL* does not require an application to explicitly reference any of the *JUEL* specific implementation classes.

Status

JUEL is considered production stable. The code is well tested (80% coverage) and feature complete.

Availability

JUEL is licensed under the Apache License 2.0.

Requirements

JUEL requires Java 5 or later.

2. JUEL Guide

This guide gives a brief introduction to *JUEL*. However, this is *not* an EL tutorial. Before using *JUEL*, we strongly recommend to get familiar with the Unified EL basics by taking a look at the specification document, which is available here.

The *JUEL* guide divides into the following sections:

1. **Quickstart** – Gets you started with *JUEL*.
2. **Basic Classes** – Covers *JUEL*'s expression factory and expression types.
3. **Utility Classes** – Introduces *JUEL*'s simple context and resolver implementations.
4. **Advanced Topics** – Talks about trees, caching, builders, extensions, etc.
5. **Specification Issues** – Known defects, clarifications, notes, ...

2.1. Getting Started

The *JUEL* distribution contains the following JAR files:

1. `juel-api-2.2.x.jar` - contains the `javax.el` API classes.
2. `juel-impl-2.2.x.jar` - contains the `de.odysseus.el` implementation classes.
3. `juel-spi-2.2.x.jar` - contains the `META-INF/service/javax.el.ExpressionFactory` service provider resource. (You will need this if you have several expression language implementations on your classpath and want to force *JUEL*s implementation to be chosen by `ExpressionFactory.newInstance()`).

Here's all you need to use the EL in your application (assuming you added the JUEL JAR files to your classpath and did `import javax.el.*`):

1. **Factory and Context**

```
// the ExpressionFactory implementation is de.odysseus.el.ExpressionFactoryImpl
ExpressionFactory factory = new de.odysseus.el.ExpressionFactoryImpl();

// package de.odysseus.el.util provides a ready-to-use subclass of ELContext
de.odysseus.el.util.SimpleContext context = new de.odysseus.el.util.SimpleContext();
```

2. **Functions and Variables**

```
// map function math:max(int, int) to java.lang.Math.max(int, int)
context.setFunction("math", "max", Math.class.getMethod("max", int.class, int.class));

// map variable foo to 0
context.setVariable("foo", factory.createValueExpression(0, int.class));
```

3. **Parse and Evaluate**

```
// parse our expression
ValueExpression e = factory.createValueExpression(context, "${math:max(foo,bar)}", int.class);

// set value for top-level property "bar" to 1
factory.createValueExpression(context, "${bar}", int.class).setValue(context, 1);

// get value for our expression
System.out.println(e.getValue(context)); // --> 1
```

2.2. Basic Classes

This section walks through the concrete classes provided by *JUEL*, that make up the core of the evaluation process: the factory and the various kinds of expressions it creates.

We do not fully cover inherited behavior, which is already described by the API super classes. Rather, we focus on additional methods provided by *JUEL* as extensions to the API as well as implementation specific information.

2.2.1. Expression Factory

To start using the EL, you need an instance of `javax.el.ExpressionFactory`. The expression factory is used to create expressions of various types.

JUEL's expression factory implementation is `de.odysseus.el.ExpressionFactoryImpl`. The easiest way to obtain an expression factory instance is

```
javax.el.ExpressionFactory factory = new de.odysseus.el.ExpressionFactoryImpl();
```

An expression factory is thread-safe and can create an unlimited number of expressions. The expression factory provides operations to

- perform type coercions,
- create tree value expressions,
- create object value expressions,
- create tree method expressions.

Expression Cache

Each factory instance uses its own expression cache. Caching expressions can be an important issue, because parsing is relative expensive. An expression cache maps expression strings to their parsed representations (trees).

JUEL provides a caching interface which allows applications to use their own caching mechanism. However, in most scenarios, *JUEL*'s default implementation (based on `java.util.concurrent.ConcurrentHashMap` and `java.util.concurrent.ConcurrentLinkedQueue`) should be fine.

> **Note**
> The caching mechanism has been rewritten for JUEL 2.2.5 to improve performance.

The default constructor uses a maximum cache size of 1000. You may specify a different value - say 5000 - by specifying the `javax.el.cacheSize` property.

```
java.util.Properties properties = new java.util.Properties();
properties.put("javax.el.cacheSize", "5000");
javax.el.ExpressionFactory factory = new de.odysseus.el.ExpressionFactoryImpl(properties);
```

Using your own caching mechanism is covered in the Advanced Topics secion.

Type Conversions

Type conversions are performed at several points while evaluating expressions.

- Operands are coerced when performing arithmetic or logical operations
- Value expression results are coerced to the expected type specified at creation time
- For literal method expressions the text is coerced to the type specified at creation time
- For non-literal method expressions the last property is coerced to a method name
- Composite expression coerce their sub-expressions to strings before concatenating them

All these coercions are done following the same rules. The specification describes these coercion rules in detail. It supports converting between string, character, boolean, enumeration and number types. Additionally, the conversion of strings to other types is supported by the use of (Java Beans) property editors. The EL makes the coercion rules available to client applications via the expression factory method

```
ExpressionFactoryImpl.coerceToType(Object, Class<?>)
```

whose return type is `Object`.

JUEL can be configured to use alternative coercion rules as described in the Advanced Topics secion.

Factory Configuration

The factory may be configured via property files. The mechanism described here is used when an expression factory is created without specifying properties. The lookup procedure for properties is as follows:

1. `JAVA_HOME/lib/el.properties` - If this file contains property `javax.el.ExpressionFactory` whose value is `de.odysseus.el.ExpressionFactoryImpl`, its properties are loaded and taken as default properties.
2. `System.getProperties()` - if the previous rule (1) did not match and system property `javax.el.ExpressionFactory` is set to `de.odysseus.el.ExpressionFactoryImpl`, the system properties are taken as default properties.
3. `el.properties` anywhere on your classpath - These properties may override the default properties from (1) or (2).

Having this, the following properties are read:

- `javax.el.cacheSize` - expression cache size (default is 1000)
- `javax.el.methodInvocations` - set to `true` to allow method invocations. Please refer to the Advanced Topics section for more on this.
- `javax.el.nullProperties` - set to `true` to resolve `null` properties. Please refer to the Advanced Topics section for more on this.
- `javax.el.varArgs` - set to `true` to allow vararg function/method calls in your expressions.

The factory class also provides constructors which let you explicitly pass your properties. If you just want to switch between JEE5 and JEE6 style, *JUEL* provides enum constants to use as profiles.

2.2.2. Value Expressions

Value expressions are expressions that are evaluated in the "classical sense". There are two kinds of value expressions: those created by parsing an expression string and those simply wrapping an object.

A `javax.el.ValueExpression` is evaluated by calling its `getValue(ELContext)` method. Value expressions can also be writable and provide methods `isReadOnly(ELContext)`, `getType(ELContext)` and `setValue(ELContext, Object)`.

A value expression is called an *lvalue expression* if its expression string is an eval expression (`#{...}` or `${...}`) consisting of a single identifier or a nonliteral prefix (function, identifier or nested expression), followed by a sequence of property operators (`.` or `[]`). All other value expressions are called *non-lvalue expressions*.

For non-lvalue expressions

- `getType(ELContext)` method will always return `null`.
- `isReadOnly(ELContext)` method will always return `true`.
- `setValue(ELContext, Object)` method will always throw an exception.

Tree Value Expressions

Creating a tree value expression involves

1. parsing an expression string and building an abstract syntax tree,
2. binding functions and variables using the mappers provided by the context.

Once created, a tree value expression can be evaluated using the `getValue(ELContext)` method. The result is automatically coerced to the expected type given at creation time.

Class `de.odysseus.el.TreeValueExpression` is a subclass of `javax.el.ValueExpression`, which is used by *JUEL* to represent a value expression, that has been created from an expression string. It is the return type of

```
ExpressionFactoryImpl.createValueExpression(ELContext, String, Class<?>)
```


In addition to the methods available for `javax.el.ValueExpression`, it provides methods

- `void dump(java.io.PrintWriter writer)` – dump parse tree
- `boolean isDeferred()` – true if expression is deferred (contains eval expressions `#{...}`)
- `boolean isLeftValue()` – true if expression is an lvalue expression.

```
import java.io.*;
import de.odysseus.el.*;
import de.odysseus.el.util.*;
...
ExpressionFactoryImpl factory = new ExpressionFactoryImpl();
SimpleContext context = new SimpleContext(); // more on this here...
TreeValueExpression e = factory.createValueExpression(context, "#{pi/2}", Object.class);
PrintWriter out = new PrintWriter(System.out);
e.dump(out);
// +- #{...}
//    |
//    +- '/'
//       |
//       +- pi
//       |
//       +- 2
out.flush();
System.out.println(e.isDeferred()); // true
System.out.println(e.isLeftValue()); // false
...
```

Object Value Expressions

An object value expression simply wraps an object giving it an "expression facade". At the first place, object expressions are used to define variables.

Once created, an object value expression can be evaluated using the `getValue(ELContext)` method, which simply returns the wrapped object, coerced to the expected type provided at creation time.

Class `de.odysseus.el.ObjectValueExpression` is a subclass of `javax.el.ValueExpression`, which is used by *JUEL* to represent a value expression, that has been created from an object. It is the return type of

```
ExpressionFactoryImpl.createValueExpression(Object, Class<?>)
```

This class provides no extra methods to those available for `javax.el.ValueExpression`.

```
import java.io.*;
import de.odysseus.el.*;
import de.odysseus.el.util.*;
...
ExpressionFactoryImpl factory = new ExpressionFactoryImpl();
SimpleContext context = new SimpleContext(); // more on this here...
ObjectValueExpression e = factory.createValueExpression(Math.PI, Double.class);
context.setVariable("pi", e);
...
```

2.2.3. Method Expressions

Method expressions can be "invoked". A `javax.el.MethodExpression` is invoked by calling its `invoke(ELContext, Object<?>[])` method. The specification also allows to treat literal text as a method expression.

A method expression is called a *literal method expression* if its underlying expression is literal text (that is, `isLiteralText()` returns `true`). All other method expressions are called *non-literal method expressions*. Non-literal method expressions share the same syntax as lvalue expressions.

For literal method expressions

- `invoke(ELContext, Object<?>[])` simply returns the expression string, optionally coerced to the expected return type specified at creation time.
- `getMethodInfo(ELContext)` always returns `null`.

On the other hand, non-literal method expressions refer to a `java.lang.reflext.Method` which can be invoked or used to create a `javax.el.MethodInfo` instance. For non-literal method expressions

- `invoke(ELContext, Object<?>[])` evaluates the expression to a `java.lang.reflext.Method` and invokes that method, passing over the given actual parameters.
- the found method must match the expected return type (if it is not `null`) and the argument types given at creation time; otherwise an exception is thrown.

Tree Method Expressions

Class `de.odysseus.el.TreeMethodExpression` is a subclass of `javax.el.MethodExpression`, which is used by *JUEL* to represent method expressions. It is the return type of

`ExpressionFactoryImpl.createMethodExpression(ELContext, String, Class<?>, Class<?>[])`

In addition to the methods declared by `javax.el.MethodExpression`, it provides

- `void dump(java.io.PrintWriter writer)` – dump parse tree
- `boolean isDeferred()` – `true` if expression is deferred (contains eval expressions `#{...}`)

```
import java.io.*;
import de.odysseus.el.*;
import de.odysseus.el.util.*;
...
ExpressionFactoryImpl factory = new ExpressionFactoryImpl();
SimpleContext context = new SimpleContext(); // more on this here...
TreeMethodExpression e =
  factory.createMethodExpression(context, "#{x.toString}", String.class, new Class[]{});
PrintWriter out = new PrintWriter(System.out);
e.dump(out);
// +- #{...}
//    |
//    +- . toString
//       |
//       +- x
out.flush();
System.out.println(e.isDeferred()); // true
...
```

2.3. Utility Classes

When creating and evaluating expressions, some other important classes come into play: a `javax.el.ELContext` is required at creation time and evaluation time. It contains methods to access a function mapper (`javax.el.FunctionMapper`), a variable mapper (`javax.el.VariableMapper`) and a resolver (`javax.el.ELResolver`).

- At creation time, the context's function mapper and variable mapper are used to bind function invocations to static methods and identifiers (variables) to value expressions. The context's resolver is *not* used at creation time.
- At evaluation time, the context's resolver is used for property resolutions and to resolve unbound identifiers (top-level properties). The context's function mapper and variable mapper are *not* used at evaluation time.

JUEL provides simple implementations of these classes to get you using the unified EL "out of the box".

2.3.1. Simple Context

Class `de.odysseus.el.util.SimpleContext` is a simple context implementation. It can be used at creation time as well as evaluation time.

For use at creation time, it provides the following methods.

- `setFunction(String prefix, String name, java.lang.reflect.Method method)` to define a method as a function for the given prefix and name. Functions without a namespace must pass in the empty string as `prefix`. The supplied method must be declared as public and static.
- `setVariable(String name, javax.el.ValueExpression expression)` to define a value expression as a variable for the given name. (This is equivalent to `getVariableMapper().setVariable(String name, javax.el.ValueExpression expression)`.)

The following example defines function `math:sin` and variable `pi` and uses them in an expression.

```
import javax.el.*;
import de.odysseus.el.util.SimpleContext;
import de.odysseus.el.ExpressionFactoryImpl;
...
ExpressionFactory factory = new ExpressionFactoryImpl();
SimpleContext context = new SimpleContext();
context.setFunction("math", "sin", Math.class.getMethod("sin", double.class));
context.setVariable("pi", factory.createValueExpression(Math.PI, double.class));
ValueExpression expr = factory.createValueExpression(context, "${math:sin(pi/2)}", double.class);
System.out.println("math:sin(pi/2) = " + expr.getValue(context)); // 1.0
```

At evaluation time, a `javax.el.ELResolver` is required for property resolution and to resolve identifiers, that have not been bound to a variable. The `getELResolver()` method is used at evaluation time to access the context's resolver instance.

A resolver may be passed to a `SimpleContext` at construction time. If the default constructor is used, calling `getELResolver()` will lazily create an instance of `de.odysseus.el.util.SimpleResolver`.

2.3.2. Simple Resolver

JUEL provides the `de.odysseus.el.util.SimpleResolver` class for use as a simple resolver, suitable to resolve top-level identifiers and to delegate to another resolver provided at construction time.

If no resolver delegate is supplied, a composite resolver will be used as default, capable of resolving bean properties, array values, list values, resource values and map values.

A resolver is made to resolve properties. It operates on a pair of objects, called *base* and *property*. *JUEL*'s simple resolver maintains a map to directly resolve top-level properties, that is `base == null`. Resolution for base/property pairs with `base != null` is delegated.

Finally, a simple resolver may also be flagged as "read-only". In this case, invoking the `setValue(ELContext, Object, Object, Object)` method will throw an exception.

```
import java.util.Date;
import javax.el.*;
import de.odysseus.el.util.SimpleContext;
import de.odysseus.el.util.SimpleResolver;
import de.odysseus.el.ExpressionFactoryImpl;
...
ExpressionFactory factory = new ExpressionFactoryImpl();
SimpleContext context = new SimpleContext(new SimpleResolver());

// resolve top-level property
factory.createValueExpression(context, "#{pi}", double.class).setValue(context, Math.PI);
ValueExpression expr1 = factory.createValueExpression(context, "${pi/2}", double.class);
System.out.println("pi/2 = " + expr1.getValue(context)); // 1.5707963...
```

```
// resolve bean property
factory.createValueExpression(context, "#{current}", Date.class).setValue(context, new Date());
ValueExpression expr2 = factory.createValueExpression(context, "${current.time}", long.class);
System.out.println("current.time = " + expr2.getValue(context));
```

2.4. Plugin JUEL

A recent addition to `javax.el.ExpressionFactory` have been the static methods `newInstance()` and `newInstance(java.util.Properties)`.

With these methods, selection of a particular EL implementation is completely transparent to the application code. E.g., the first line of our Quickstart example could be rewritten as

```
ExpressionFactory factory = ExpressionFactory.newInstance();
```

Either of the new methods will determine a factory implementation class and create an instance of it. The first variant will use its default constructor. The latter will use a constructor taking a single `java.util.Properties` object as parameter. The lookup procedure uses the Service Provider API as detailed in the JAR specification.

The `juel-spi-2.2.x.jar` does the trick: if on your classpath, the lookup procedure will detect `de.odysseus.el.ExpressionFactoryImpl` as service provider class.

This way, *JUEL* can be used without code references to any of its implementation specific classes. Just `javax.el.*`...

> **Note**
>
> The new API is part of the EL since version 2.2 (JEE6). Therefore, it may not be supported in environments which are based on EL 2.1 (JEE5).

Depending on your application's scenario, there may be several ways to register *JUEL* as default EL implementation.

- Place the JUEL JARs into directory `JRE_HOME/lib/ext`. This will make JUEL available globally for all applications running in that environment.
- You may simply drop `juel-impl-2.2.x.jar` and `juel-spi-2.2.x.jar` into your `/WEB-INF/lib` directory. This will result in using *JUEL* as EL implementation for that particular web application.
- Of course you can also add the jar files to your classpath manually.

Please refer to the section on Factory Configuration on how to configure an expression factory via property files.

2.5. Advanced Topics

This section covers some advanced *JUEL* topics.

Expression Trees

An expression tree refers to the parsed representation of an expression string. The basic classes and interfaces related to expression trees are contained in package `de.odysseus.el.tree`. We won't cover all the tree related classes here. Rather, we focus on the classes that can be used to provide a customized tree cache and builder.

1. `Tree` – This class represents a parsed expression string.
2. `TreeBuilder` – General interface containing a single `build(String)` method. A tree builder must be thread safe. The default implementation is `de.odysseus.el.tree.impl.Builder`.
3. `TreeCache` – General interface containing methods `get(String)` and `put(String, Tree)`. A tree cache must be thread safe, too. The default implementation is `de.odysseus.el.tree.impl.Cache`.
4. `TreeStore` – This class just combines a builder and a cache and contains a single `get(String)` method.

The expression factory uses its tree store to create tree expressions. The factory class provides a constructor which takes a tree store as an argument.

Using a customized Builder

It should be noted that one could write a builder by implementing the `de.odysseus.el.tree.TreeBuilder` interface from scratch. However, you may also want to subclass the `Builder` class and override the `createParser()` to work with a modified parser implementation.

Either way, the new tree builder can be passed to a factory via

```
TreeStore store = new TreeStore(new MyBuilder(), new Cache(100));
ExpressionFactory factory = new ExpressionFactoryImpl(store);
```

As an alternative, you may set property

```
de.odysseus.el.tree.TreeBuilder
```

to the fully qualified class name of your builder implementation.

Enabling/Disabling Method Invocations

Many people have noticed the lack of method invocations as a major weakness of the unified expression language. When talking about method invocations, we mean expressions like `${foo.matches('[0-9]+')}` that aren't supported by the 2.1 standard. However, method invocations have been added in maintenance release 2 of JSR 245, which is supported by *JUEL*.

> **Warning**
> *JUEL's* proprietary API for method invocations has been removed in version 2.2.

To enable (disable) expressions using method invocations, you may set property

```
javax.el.methodInvocations
```

to `true` (`false`).

Method invocations are enabled in profile *JEE6* (default) and disabled in *JEE5*.

Enabling/Disabling VarArgs

The EL specification does not support function calls with variable argument lists. That is, if we bind `String.format(String, Object...)` to function `str:format`, the expression `${str:format('Hey %s','Joe')}` will not work.

To enable (disable) VarArgs in function and method invocations, you may set property

```
javax.el.varArgs
```

to `true` (`false`).

VarArg invocations are enabled in profile *JEE6* (default) and disabled in *JEE5*.

Enabling/Disabling null Properties

The EL specification describes the evaluation semantics of `base[property]`. If `property` is `null`, the specification states not to resolve `null` on `base`. Rather, `null` should be returned if `getValue(...)` has been called and a `PropertyNotFoundException` should be thrown else. As a consequence, it is impossible to resolve `null` as a key in a map. However, *JUEL*'s expression factory may be configured to resolve `null` like any other property value.

To enable (disable) `null` as an EL property value, you may set property

```
javax.el.nullProperties
```

to `true` (`false`).

Assume that identifier `map` resolves to a `java.util.Map`.

- If feature `javax.el.nullProperties` has been disabled, evaluating `${base[null]}` as an rvalue (lvalue) will return `null` (throw an exception).
- If feature `javax.el.nullProperties` has been enabled, evaluating `${base[null]}` as an rvalue (lvalue) will get (put) the value for key `null` in that map.

The default is not to allow `null` as an EL property value.

Enabling/Disabling ignoring of expected return type

The EL specification allows to determine an expected return type for method expressions. The return type should then be checked to match the actual return type of the method to invoke. Unfortunately, the EL reference implementation ignores this parameter completely. This caused some "legacy" code to not recognize that they're passing wrong types. When switching to JUEL as their EL implementation, this causes an exception to be thrown.

For compatibility, JUEL lets you choose to ignore the expected return type passed to `EpressionFactory.createMethodExpression()` when looking up a method to invoke. (This option has no effect when evaluating literal method expressions, where the expected return type acts as coercion target type.)

To enable (disable) ignoring of the expected return type parameter, you may set property

```
javax.el.ignoreReturnType
```

to `true` (`false`).

The default is respect (i.e. *not* to ignore) the expected return type parameter.

Using a customized Cache

The default lru cache implementation can be customized by specifying a maximum cache size. However, it might be desired to use a different caching mechanism. Doing this means to provide a class that implements the `TreeCache` interface.

Now, having a new cache implementatation, it can be passed to a factoy via

```
TreeStore store = new TreeStore(new Builder(), new MyCache());
ExpressionFactory factory = new ExpressionFactoryImpl(store);
```

Tree Expressions

In the basics section, we already presented the `TreeValueExpression` and `TreeMethodExpression` classes, which are used to represent parsed expressions.

Equality

As for all objects, the `equals(Object)` method is used to test for equality. The specification notes that two expressions of the same type are equal if and only if they have an identical parsed representation.

This makes clear, that the expression string cannot serve as a sufficient condition for equality testing. Consider expression string `${foo}`. When creating tree expressions from that string using different variable mappings for `foo`, these expressions must not be considered equal. Similar, an expression string using function invocations may be used to create tree expressions with different function mappings. Even worse, `${foo()}` and `${bar()}` may be equal if `foo` and `bar` referred to the same method at creation time.

To handle these requirements, *JUEL* separates the variable and function bindings from the pure parse tree. The tree only depends on the expression string and can therefore be reused by all expressions created from a string. The bindings are then created from the tree, variable mapper and function mapper. Together, the tree and bindings form the core of a tree expression.

When comparing tree expressions, the trees are structurally compared, ignoring the names of functions and variables. Instead, the corresponding methods and value expressions bound to them are compared.

Serialization

As required by the specification, all expressions have to be serializable. When serializing a tree expression, the expression string is serialized, not the tree. On deserialization, the tree is rebuilt from the expression string.

Customizing Type Conversions

The rules to apply when coercing objects is described in the specification. However, in a non-JEE environment, it might be desired to apply application-specific or additional type conversions. To do this, you must provide *JUEL*'s expression factory with an implementation of

```
de.odysseus.el.misc.TypeConverter
```

which defines a single method:

```
public <T> T convert(Object value, Class<T> type) throws ELException
```

The default converter is implemented in `de.odysseus.el.misc.TypeConverterImpl`. To use your new type converter, pass an instance of it to the factory constructor

```
ExpressionFactoryImpl(TreeStore store, TypeConverter converter)
```

As an alternative, you may set property

```
de.odysseus.el.misc.TypeConverter
```

to the fully qualified name of your converter class (requires your class to provide a default constructor).

The `BeanELResolver.invoke(...)` method needs type conversions to convert actual method parameters to the method's formal parameter types. Unfortunately, the resolver API doesn't provide access the an `ExpressionFactory` to use our customized conversions via `ExpressionFactory.coerceToType(...)`. *JUEL*'s bean resolver implementation consults the `javax.el.ExpressionFactory` context property to get a factory before creating a default using `ExpressionFactory.getInstance()`. That is, if you're using *JUEL*'s EL API, you may do

```
elContext.putContext(javax.el.ExpressionFactory.class, factory)
```

to make your customized type conversions available to the resolver.

2.6. Specification Issues

JUEL tries to be as close as possible to the EL specification. However, the spec isn't always clear, leaves some details open and sometimes even seems to be incorrect. For these certain gaps, *JUEL* had to make decisions that could not be derived from the specification. We still hope that the specification could be updated to make things more clear. Until then, we will have this section to list the specification issues.

1. In section 1.19, "Collected Syntax", the specification defines Nonterminal `LValueInner` (which describes an lvalue expression) as

   ```
   LValueInner ::= Identifier | NonLiteralValuePrefix (ValueSuffix)*
   ```

   This would mean - since `NonLiteralValuePrefix` can be expanded to a nested expression or function invocation - that `${(1)}` or `${foo()}` were lvalue expressions. *JUEL* considers this to be a bug and guesses that the above should read as

   ```
   LValueInner ::= Identifier | NonLiteralValuePrefix (ValueSuffix)+
   ```

   instead.

2. In section 1.2.3, "Literal Expressions", the specification states that "the escape characters \$ and \# can be used to escape what would otherwise be treated as an eval-expression". The specification doesn't

tell us if `'\'` is used to escape other characters in literal expressions, too. Consequently, *JUEL* treats `'\'` as escape character only when immediately followed by `'${'` and `'#{'`.

> **Note**
> Expression `\\${` evaluates to `'\${'`, whereas `\$` evaluates to `'\$'` and `\\` evaluates to `'\\'`.

3. In section 1.3, "Literals", the specification states that "Quotes only need to be escaped in a string value enclosed in the same type of quote". This suggests that, e.g., "You could escape a single quote in a double-quoted string, but it's not necessary". *JUEL* guesses that you can't and that the above should read as "Quotes can only be escaped in a string value enclosed in the same type of quote".

> **Note**
> The '\' in expression `${'\"'}` doesn't escape the double quote.

4. From section 1.2.1.2, "Eval-expressions as method expressions", it is clear that a single identifier expression, e.g. `${foo}`, can be used as a method expression. However, the specification doesn't tell *how* to evaluate such a method expression! Unfortunately, there's no obvious guess, here... *JUEL* evaluates method expression `${foo}` as follows (let `paramTypes` be the supplied expected method parameter types, `returnType` the expected return type):
   - Evaluate `${foo}` as a value expression
   - If the result is an instance of `java.lang.reflect.Method`
     - if the method is not static, then error.
     - if the method's parameter types do not match the `paramTypes`, then error.
     - if `returnType` is not `null` and the method's return type does not match `returnType`, then error.
     - If `MethodExpression.invoke(...)` was called, invoke the found method with the parameters passed to the `invoke` method.
     - If `MethodExpression.getMethodInfo(...)` was called, construct and return a new `MethodInfo` object.
   - *JUEL 2.2.6 and later:* If the result is an instance of `javax.el.MethodExpression`
     - If `MethodExpression.invoke(...)` was called, delegate to `invoke(...)` on the found method expression.
     - If `MethodExpression.getMethodInfo(...)` was called, delegate to `getMethodInfo(...)` on the found method expression.
   - Otherwise, error
5. Section 1.6, "Operators [] and .", describes the semantics of `base[property]`. If `property` is `null`, the speciification states not to resolve `null` on `base`. Rather, `null` should be returned if `getValue(...)` has been called and a `PropertyNotFoundException` should be thrown else. As a consequence, it would not be possible to resolve `null` as a key in a map. We think that this restriction is not really wanted and more generally, that `property == null` should not even have been treated as a special case. We have made an enhancement request, hoping that this will be changed in the future. Because this has been explicitly stated in the spec, we had to implement it this way. However, *JUEL*'s builder supports a feature `NULL_PROPERTIES` to let you resolve `null` like any other property value.

> **Note**
> Assume that `map` resolves to a `java.util.Map`. Further assume that feature `NULL_PROPERTIES` is enabled. Evaluating `${base[null]}` as an rvalue (lvalue) will get (put) the value for key `null` in that map.

6. Section 1.19, "Collected Syntax" defines Nonterminal `IntegerLiteral` to be an unsigned integer constant. Then it is said that "The value of an `IntegerLiteral` ranges from `Long.MIN_VALUE` to `Long.MAX_VALUE`". We take that as a hint that the spec wants us to parse integer literals into `Long` values. However, the positive number `|Long.MIN_VALUE|` cannot be stored in a `Long` since `Long.MAX_VALUE = |Long.MIN_VALUE| - 1`. We think that the specification should have said that "The value of an

IntegerLiteral ranges from 0 to Long.MAX_VALUE". Consequently, *JUEL* rejects |Long.MIN_VALUE| = 9223372036854775808 as an illegal integer literal.

7. Section 1.19, "Collected Syntax" defines Nonterminal FloatingFointLiteral to be an unsigned floating point constant. Then it is said that "The value of a FloatingPointLiteral ranges from Double.MIN_VALUE to Double.MAX_VALUE". We take that as a hint that the spec wants us to parse floating point literals into Double values. However, since Double.MIN_VALUE is the smallest positive value that can be stored in a Double, this would exclude zero from the range of valid floating point constants! We think that the specification should have said that "The value of a FloatingPointLiteral ranges from 0 to Double.MAX_VALUE". Consequently, *JUEL* accepts 0.0 as a legal floating point literal.

3. JUEL Project

3.1. History of Changes

Version 2.2.7 (2014/02/06)

developer: **cbe** ***type:*** **update** ***fixes:*** **73**

updated method invocation code to use Method.setAccessible() only if necessary.

developer: **cbe** ***context:*** **build** ***type:*** **add** ***fixes:*** **79**

Catch SecurityException when accessing default el.properties.

Version 2.2.6 (2013/01/11)

developer: **cbe** ***type:*** **update** ***thanks to:*** **Oleg Varaksin, Arjan Tijms.** ***fixes:*** **71**

For a single identifier method expression, if the identifier evaluates to another method expression, invoke that method expression. (The semantics for this case is not covered in the specification. See also *this issue*. This change was made to improve compatibility with other EL implementations.)

developer: **cbe** ***context:*** **build** ***type:*** **add** ***thanks to:*** **Adam Crume.**

Added OSGi bundle manifest entries for juel-api and juel-impl and declared de.odysseus.el.ExpressionFactoryImpl as an OSGi service.

Version 2.2.5 (2012/07/08)

developer: **cbe** ***context:*** **build** ***type:*** **update** ***thanks to:*** **Oleg Varaksin.** ***fixes:*** **3521406**

Changed scope for dependency from juel-impl to juel-api to provided. Therefore, to include juel-api, users now need to explicitly add it as a dependency.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Re-implemented Cache based on ConcurrentHashMap and ConcurrentLinkedQueue to improve performance in multi-threaded environments.

developer: **cbe** ***context:*** **code** ***type:*** **update** ***thanks to:*** **Martin Koci.** ***fixes:*** **3529970**

Determine target type and coerce argument in ExpressionNode.setValue().

developer: **cbe** ***context:*** **code** ***type:*** **fix** ***thanks to:*** **Martin Koci.** ***fixes:*** **834616**

Catch and re-throw IllegalArgumentException as ELException in BeanELResolver.setValue().

developer: **cbe** ***context:*** **code** ***type:*** **update**

Updated messages with keys error.coerce.type and error.coerce.value to include value that could not be coerced.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Create HashMap in `ELContext` lazily.

***developer:* cbe *context:* code *type:* update**

`BeanELResolver` now caches property access methods.

***developer:* cbe *context:* code *type:* fix *fixes:* 3420591**

`ELContext.getContext(key)` and `putContext(key, value)` should throw NPE if key is `null`

Version 2.2.4 (2011/09/30)

***developer:* cbe *context:* code *type:* add**

Added property `javax.el.ignoreReturnType` to ignore the expected return type passed to `EpressionFactory.createMethodExpression()` when looking up a method to invoke. (This option has no effect when evaluating literal method expressions, where the expected return type acts as coercion target type.)

Version 2.2.3 (2011/01/30)

***developer:* cbe *context:* code *type:* update *thanks to:* Martin Koci. *fixes:* 3154206**

fixed: a method invocation node invoked via `ValueExpression.getValue(...)` should return `null` if its base (prefix) expression evaluates to `null`.

***developer:* cbe *context:* build *type:* update**

Moved parent `pom.xml` from `modules` to project root to simplify maven release build.

Version 2.2.2 (2010/11/12)

***developer:* cbe *context:* build *type:* update**

Updated POMs to sync JUEL releases to maven central via Sonatype's repository hosting service.

***developer:* cbe *context:* code *type:* update *fixes:* 3104608**

fixed: `MethodExpression.invoke(...)` should ignore passed parameter types if arguments are specified in the expression as in `${foo.bar(123)}`.

***developer:* cbe *context:* code *type:* update *fixes:* 3095122**

Improved method lookup in method expressions (when calling `MethodExpression.invoke(...)`, `Method.setAccessible(true)` is used if necessary).

***developer:* cbe *context:* code *type:* update**

Changed implementation of `ASTProperty.getValueReference()` to throw an exception if `base` expression evaluates to `null`. Updated documentation of `ValueExpression.getValueReference()` accordingly.

Version 2.2.1 (2009/12/13)

***developer:* cbe *context:* code *type:* add**

Added `MethodExpression.isParametersProvided()`. This method was added silently to the API (wasn't mentioned anywhere prior to the final release of EE6) and forces this release...

***developer:* cbe *context:* build *type:* update**

Added OSGi attributes to manifest files of `juel-api-2.2.x.jar`, `juel-impl-2.2.x.jar` and `juel-spi-2.2.x.jar`. E.g. the latter two can now be added as OSGi modules to glassfish v3.

***developer:* cbe *context:* code *type:* add**

Added `SimpleResolver.setELResolver(ELResolver)`.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Removed static field `BeanELResolver.DEFAULT_FACTORY`. Lazily get a factory for type conversions when needed to coerce method parameters instead.

Version 2.2.0 (2009/12/01)

developer: **cbe** ***context:*** **code** ***type:*** **update**

Improved implementation of type conversions in `de.odysseus.el.misc.TypeConverterImpl`.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added some more unit tests.

Version 2.2.0-rc3 (2009/11/08)

developer: **cbe** ***context:*** **code** ***type:*** **update**

If no properties are available in *JUEL*'s expression factory (neither passed in nor from `el.properties`), use system properties if `System.getProperty("javax.el.ExpressionFactory")` points to *JUEL*.

developer: **cbe** ***context:*** **admin** ***type:*** **update**

Split code base into modules `modules/api`, `modules/impl` and `modules/spi`; added maven build for these.

developer: **cbe** ***context:*** **build** ***type:*** **add**

Separated the JAR service provider (`META-INF/services/javax.el.ExpressionFactory`) into its own `juel-spi-2.2.x.jar`. This allows to have `juel-impl-2.2.x.jar` on your classpath without forcing *JUEL* to be used by `ExpressionFactory.newInstance()`.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added `ExpressionFactoryImpl.Profile` enum type and several new constructors to easily choose between *JEE5* and *JEE6* (default) behavior.

developer: **cbe** ***context:*** **docs** ***type:*** **update**

Updated documentation for 2.2.

Version 2.1.3/2.2.0-rc2 (2009/10/09)

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added `ExpressionFactoryImpl(TreeStore store, TypeConverter converter)` constructor.

developer: **cbe** ***context:*** **code** ***type:*** **fix** ***thanks to:*** **Pavel Vojtechovsky.** ***fixes:*** **2871773**

Fixed: `ListELResolver.getValue(...)` should return `null` when index is out of range.

developer: **cbe** ***context:*** **code** ***type:*** **fix** ***thanks to:*** **Pavel Vojtechovsky.** ***fixes:*** **2871795**

Fixed: second operand is always evaluated in and/or operations (e.g. `${true or false}`).

developer: **cbe** ***context:*** **code** ***type:*** **fix** ***fixes:*** **2822943**

Fixed: `BeanELResolver` should use `Method.setAccessible(true)`.

developer: **cbe** ***context:*** **docs** ***type:*** **update**

Updated copyright notes.

Version 2.2.0-rc1 (2009/08/09)

developer: **cbe** ***context:*** **code** ***type:*** **add**

Implemented changes from JSR 245, maintenance release 2. This replaces *JUEL*s proprietary API for method invocations. The `BeanELResolver` class now provides a generic default for method invocations using reflection. Method invocations are disabled/enabled via the `javax.el.methodInvocations` property.

Version 2.1.2 (2009/04/26)

***developer:* cbe *context:* code *type:* fix *thanks to:* Gerhard Petracek. *fixes:* 2748538**

Fixed: for a single identifier expression `${foo}`, `ValueExpression.setValue(...)` always calls `ELContext.setValue(...)`, even if `foo` is bound to a variable.

***developer:* cbe *context:* build *type:* update**

Renamed API/impl jars to `juel-api-<version>.jar` and `juel-impl-<version>.jar` to reflect maven artifact names `juel-api` and `juel-impl`.

Version 2.1.1 (2009/03/21)

***developer:* cbe *context:* code *type:* add**

Added support for bracket operator in method invocations (e.g. `${foo[bar](baz)}`).

***developer:* cbe *context:* code *type:* add**

Added implementation of `javax.el` api classes.

***developer:* cbe *context:* build *type:* update *thanks to:* Wolfgang Häfelinger.**

Updated `pom.xml` to package the following jar files: `juel-<version>.jar`, `juel-<version>-api.jar` and `juel-<version>-impl.jar`.

***developer:* cbe *context:* code *type:* fix *thanks to:* Adam Winer. *fixes:* 2590830**

Fixed: `SimpleContext.getVariableMapper()` returns `null`

***developer:* cbe *context:* code *type:* add**

Experimental support for syntax extensions.

***developer:* cbe *context:* build *type:* update**

Reorganized project structure to meet maven's standard layout.

***developer:* cbe *context:* code *type:* update**

Improved method invocation support by passing `de.odysseus.el.misc.MethodInvocation` as property to `ELResolver.getValue(..., Object property)`.

***developer:* cbe *context:* code *type:* add**

Added support for varargs.

***developer:* cbe *context:* code *type:* add**

Introduce `TypeConverter` to allow for customized coercion rules.

***developer:* cbe *context:* code *type:* remove**

Removed old `TypeConversions` (use `TypeConverter.DEFAULT`).

***developer:* cbe *context:* docs *type:* update**

Update to fop-0.95-beta and forrester-0.3.3.

***developer:* cbe *context:* admin *type:* update**

Moved to SVN.

Version 2.1.0 (2007/03/06)

developer: **cbe** ***context:*** **code** ***type:*** **update**

Use `StringBuilder` instead of `StringBuffer` (performance).

developer: **cbe** ***context:*** **code** ***type:*** **update**

Update API sources from glassfish.

Version 2.1.0-rc3 (2006/10/20)

developer: **cbe** ***context:*** **code** ***type:*** **fix** ***thanks to:*** **Frédéric Esnault.**

`ListELResolver` was missing in `SimpleResolver`'s default chain of resolver delegates.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Update API sources from glassfish.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Minor performance improvements in type conversions and number operations.

Version 2.1.0-rc2 (2006/10/06)

developer: **cbe** ***context:*** **code** ***type:*** **update**

Relaxed matching of return type for nonliteral `MethodExpression`'s. The actual method return type is checked be assignable to the expression's expected return type.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Let `ExpressionFactory`'s default constructor read properties from `el.properties`.

developer: **cbe** ***context:*** **admin** ***type:*** **update**

Updated API classes to include new API methods `ExpressionFactory.newInstance()` and `ExpressionFactory.newInstance(java.util.Properties)`.

developer: **cbe** ***context:*** **build** ***type:*** **add**

Package Jars with `META-INF/services/javax.el.ExpressionFactory` to register *JUEL* as EL service provider.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added `Builder.Feature.NULL_PROPERTIES` to resolve `${map[null]}`.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Generified `TypeConversions.coerceToEnum(...)` and `TypeConversions.coerceToEnum(...)`.

developer: **cbe** ***context:*** **code** ***type:*** **fix**

Coerce `null` function parameters whose type is primitive.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Minor scanner cleanup.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Increased default cache size to 1000.

developer: **cbe** ***context:*** **code** ***type:*** **update**

`ExpressionFactoryImpl` no longer `final` to allow customization by subclassing. E.g. using *JUEL* with JSF requires calling a default constructor.

Version 2.1.0-rc1 (2006/07/18)

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added support for method invocations as in `${foo.bar(1)}` (disabled by default).

developer: **cbe** ***context:*** **code** ***type:*** **fix**

Reject identifier `instanceof`.

developer: **cbe** ***context:*** **docs** ***type:*** **add**

Added "Advanced Topics" section.

developer: **cbe** ***context:*** **code** ***type:*** **remove**

Removed support for system property `de.odysseus.el.factory.builder`.

developer: **cbe** ***context:*** **design** ***type:*** **update**

Moved default tree cache implementation to package `de.odysseus.el.tree.impl`.

developer: **cbe** ***context:*** **design** ***type:*** **update**

Moved node implementation classes to package `de.odysseus.el.tree.impl.ast`.

developer: **cbe** ***context:*** **code** ***type:*** **remove**

Removed deprecated methods from `SimpleResolver`.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Do not coerce `null` function parameters.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Minor improvemnents in `BooleanOperations` and `TypeConversions`.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Replaced JFlex scanner by handcoded scanner.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Lazy initialize parser's lookahead token list.

Version 2.1.0-b2 (2006/07/01)

developer: **cbe** ***context:*** **docs** ***type:*** **add**

Added specification issues on number literals.

developer: **cbe** ***context:*** **code** ***type:*** **remove**

Finally removed the old JavaCC parser.

developer: **cbe** ***context:*** **docs** ***type:*** **add**

Added some more Javadocs.

developer: **cbe** ***context:*** **code** ***type:*** **fix**

Avoid `NumberFormatException` when parsing integer/floating point literals.

developer: **cbe** ***context:*** **code** ***type:*** **remove**

Removed static`TreeBuilder.DEFAULT` constant.

developer: **cbe** ***context:*** **code** ***type:*** **fix**

Take builder and expected type into account when comparing tree expressions.

Version 2.1.0-b1 (2006/06/18)

developer: **cbe** ***context:*** **docs** ***type:*** **add**

Added documentation (HTML and PDF).

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added `TreeValueExpression.isLeftValue()`.

developer: **cbe** ***context:*** **code** ***type:*** **remove**

Removed `ExpressionNode.isLiteralValue()`.

developer: **cbe** ***context:*** **build** ***type:*** **add**

Added more jar manifest attributes.

developer: **cbe** ***context:*** **build** ***type:*** **update**

Let javac include line and source debug information.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added secondary cache (`WeakHashMap`) to `TreeCache.Default`.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Lazy initialize `SimpleContext.ELResolver`.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Configure default builder class via system property `de.odysseus.el.factory.builder`.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Added @Override annotations.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added SAX XML filter sample.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Simplified `SimpleResolver` (now only handles top-level properties) .

developer: **cbe** ***context:*** **code** ***type:*** **update**

Deprecated `SimpleContext.setValue(...)` and `SimpleContext.setFunctions(...)`. These methods will be removed in 2.1.0.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Lots of minor refactorings.

Version 2.1.0-a3 (2006/06/04)

developer: **cbe** ***context:*** **code** ***type:*** **fix**

Re-throw `NumberFormatException` in number coercion as `ELException`.

developer: **cbe** ***context:*** **code** ***type:*** **fix**

Expected type now mandatory for value expressions.

developer: **cbe** ***context:*** **docs** ***type:*** **add**

Added SourceForge logo to *JUEL* home page.

developer: **cbe** ***context:*** **code** ***type:*** **add**

Added a calculator sample.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Now use a new hand crafted top-down parser and a JFlex generated scanner. *This almost doubles parsing performance!*

developer: **cbe** ***context:*** **code** ***type:*** **update**

Moved the Javacc parser to package `de.odysseus.el.tree.impl.javacc`. By default, it is excluded from the *JUEL* jar file.

Version 2.1.0-a2 (2006/06/01)

developer: **cbe** ***context:*** **code** ***type:*** **update**

Include EL api sources from glassfish now (the tomcat6 code was too buggy). The sources are available under Sun's CDDL and are redistributed here. Also added a note on that in the README.txt file.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Use pure Javacc parser. We no longer use the JJTree preprocessor. The AST classes are now Javacc independent and could easily be reused with other parser generators.

developer: **cbe** ***context:*** **code** ***type:*** **update**

Improved unit tests

developer: **cbe** ***context:*** **docs** ***type:*** **add**

Added some documentation

developer: **cbe** ***context:*** **code** ***type:*** **update**

Improved parse exception formatting

Version 2.1.0-a1 (2006/05/13)

developer: **cbe** ***context:*** **admin** ***type:*** **add**

Initial Release