

Oracle ADFmc EL

Background

Oracle ADFmc is a lightweight implementation of the Oracle ADF runtime, enabling declaratively developed, native (non-browser based), applications for mobile devices on which the ADFmc runtime is present.

A key component in an ADFmc targeted application is the use of Java Expression Language ([JSR-245](#)) expressions to bind user interface components to model data.

Early in the course of development, the [JUEL project](#) was selected as the basis for the ADFmc Expression Language engine as it uses a high performance parser, and is lightweight and efficient.

However, the JUEL implementation was not a drop-in solution as it targets the Java SE platform, while ADFmc must target the Java ME platform using the Connected Limited Device Configuration as a lowest common denominator, so certain language features and libraries used in the original implementation would not be available on the targeted platforms.

In the ADFmc implementation, code dependent on unsupported Java SE elements has been modified or removed to be compatible with a Java ME, CLDC target platform.

Building

The source code for the Oracle ADFmc EL implementation is organized into different source folders - "core", "adfmc_util_dependencies" and "unconverted". Both the "core" and "adfmc_util_dependencies" folders must be included in the source path to build successfully. The "core" folder contains the modified JUEL source files. The "adfmc_util_dependencies" folder contains source files for some additional types or subsections of types that implement Java SE functionality that could not be easily substituted or removed from the core implementation, which must be built with the core source files.

No other libraries, apart from the java runtime libraries themselves, are necessary on the classpath for building.

Note that the source files in the "unconverted" folder are also available for use, but have not been ported to be compatible with Java ME targets and will not build without modification.

Regarding javadoc

Substantial changes have been made to the original JUEL implementation, and while the code has been altered, in many cases the javadoc has not. The behavior described by the javadoc on a given type or its members may or may not be accurate. Check the implementation for [changes](#) before assuming that the javadoc is still accurate.

Compatibility and changes

The original JUEL implementation deviates from the [EL specification](#) in some areas. These changes are described [here](#). In addition to these differences, further behavioral and structural changes have been made in this implementation for Java ME compatibility, performance, and specific ADFmc design requirements.

Changes to the original source code have been [tagged](#) with an indicator to the general nature of the change. The full set of changes to the source is extensive, but many of these involve low level implementation changes for Java ME compatibility purposes and should not change the expected behavior of a functional area. There are however, some areas that have changed functionally or structurally that are noteworthy.

- Conversion of ELResolver from an abstract class to an interface

This change was made so that any java class that is rooted in another class hierarchy can implement the interface and resolve a property on itself. This interface is used in the implementation of `oracle.adfmc.impl.SelfELResolver`, which checks if the 'base' object implements the interface and delegates the ELResolver call to it, passing along the 'base' object and the 'property' string. This is used to resolve property strings on a base object that need to dispatch to a method or an accessor call without using reflection. The idea is that the implementing base object switches on the property strings passed to it in its implementation of `getValue` or `setValue`, routing to the appropriate methods. If reflection were available to the java runtime, a `BeanResolver` could generally be used for this purpose, but on a Java ME system, the string-to-call dispatching must be done explicitly.

- Type conversions

Some of the built in type conversions in the original JUEL implementation are to or from types that are not supported on the Java ME platform and have been disabled. These include `java.math.BigInteger`, `java.math.BigDecimal` and `java.lang.Enum`.

Conversion of strings to arbitrary types other than the basic `java.lang.*` types are not supported. The original JUEL implementation uses the `java.beans.PropertyEditor` and `PropertyEditorManager` classes to perform conversions of strings to values of a type passed in as a parameter, these classes are not supported on the Java ME platform.

- Static ExpressionFactory instantiation

<dont forget to change factory strings in public version>

In the original implementation, the base `ExpressionFactory` class defines a `newInstance` method which uses an ordered lookup procedure to determine the concrete implementation class to create. The implementation relies on types unsupported on the Java ME platform and has been simplified to use a private static member (`FACTORY_IMPLEMENTOR_CLASSNAME`) to specify the name of the implementation class to create. This does mean that creation is no longer dynamically configurable, which was not a requirement for ADFmc, but a mechanism for this can be re-implemented if needed.

A singleton accessor that uses the same logic has also been added to the base `ELContext` class (`getInstance()`). The value of the static member (`CONTEXT_IMPLEMENTOR_CLASSNAME`) is the concrete `ELContext` that is created when the `getInstance` method is called.

- Addition of notifications

A basic notification infrastructure has been added to `Expression` objects to support the ADFmc UI framework. Listeners can register on an `Expression` object at runtime for notifications when the value of the object the expression is bound to (or any of the objects bound to in a compound expression) changes.

This is helpful when running in a rich client environment when the model is in the same process as the UI and needs to be continuously rendered as opposed to a browser UI that is rendered or refreshed more on a request/response cycle. Instead of re-evaluating many expressions whose values may not have changed, the UI can just evaluate and re-render the content of the ones it needs to on demand.

Two notification interfaces are defined, one version that passes a generic object back as the event source argument and one version that passes an `Expression` object back as the source. `Expression` objects listen on their bound objects for the former (which must be implemented on any bound object for notifications to work) and rebroadcast in terms of the latter (passing themselves as the source). So consumers of EL expressions will use the `Expression` typed notification interfaces, and objects that can be bound to through EL expressions will source the generic notifications.

The notification interfaces and supporting classes are defined in the `oracle.adfmvc.el.event` package and sub-packages. The propagation of listener registration and notification is implemented through the various tree node classes, so there is a core dependency on these interfaces, but their use at runtime is optional.

- Variable identifier resolution and Expression caching

In the original JUEL implementation, on construction an `Expression` object binds variable identifiers in its parsed expression tree to aliased `Expression` objects through a call to `Tree.bind()`, which resolves variables through the `ELContext`'s `VariableMapper` and stores them in a `Bindings` object, which is used in subsequent evaluations to map a variable identifier to the resolved `Expression`. Note that the actual resolution through the `VariableMapper` only occurs once in the call to `Tree.bind()`, the `Bindings` object acts as an associative container for the variable name and the `Expression` object and does not try to re-resolve it on subsequent evaluations of the expression.

In the original implementation, the parsed tree for a given EL expression string is cached, but not the owning `Expression` object. So each time an `Expression` object is retrieved from the `ExpressionFactory`, a new one is created and its variables are effectively re-bound, so it will accurately reflect if a variable has been set to alias a new `Expression` through the `VariableMapper`.

This does not work well in the ADFmc implementation, however, because the Expression objects themselves are cached for a given EL expression string, so that the same instance is the event source for multiple instances of the same EL expression string. This means that the resolved alias Expression in the Bindings object could become stale while the owning Expression object is still being used, if the variable is set to another Expression through the ELContext's VariableMapper. To correct for this, the code in AstIdentifier evaluation has changed to resolve the variable through the ELContext's VariableMapper every time, as opposed to getting it from the Bindings object which may have a stale value for the aliased expression.

If it is desirable to use the original ExpressionFactory and tree-only caching implementation, use of the Bindings object at identifier evaluation time can be re-enabled as well.

Comments tagging modified source

When source files have been modified, the original source lines have been kept commented above the new implementation, and the commented source has been tagged with a comment that describes the reason for and/or the nature of the change.

Ex.

```
// Mobile: substituted type
//private final List<AstNode> nodes;
private final Vector nodes;

// Mobile: unsupported language feature (no autoboxing)
//throw new ScanException(position, "unterminated string",
quote);
throw new ScanException(position, "unterminated string", new
Character(quote));
```

All of these 'tag' comments are prefixed with "Mobile: ", so searching the source files for "Mobile:" is an easy way to find all of the source changes from the original JUEL implementation by category.

Tag comment types:

- Mobile: unsupported type

Used anywhere an unsupported Java SE type is referenced. Code depending on the type will either be changed to use an equivalent or compatible type, or removed.

- Mobile: unsupported type dependency

Generally used when a block of code or a call to a method whose implementation largely relies on an unsupported type or language feature is removed. Can also be used when a method signature is changed.

- Mobile: unsupported method

Used anywhere an unsupported Java SE method is referenced. Some Java ME types only support a subset of methods on the corresponding Java SE types. Code depending on the method will either be changed to use an equivalent or compatible operation (or set of operations), or removed.

- Mobile: unsupported language feature

Used anywhere code relies on a language feature or construct unsupported by Java ME or the javac 1.4 compiler. Examples of these include Generics, enum types, for-each loops, autoboxing and reflection. Code depending on these will either be changed to use an equivalent or compatible construct, or removed.

- Mobile: substituted type

Used anywhere an unsupported Java SE type has been replaced with an equivalent or compatible type that is available in Java ME.

- Mobile: changed signature

Used anywhere a method or type signature has been changed. This can mean a change in the type and number of parameters in a method, a change in the 'throws' clause of a method, or a change in the type a class extends or implements. This is usually due to the use of an unsupported type somewhere in the signature.

- Mobile: added type

Used on a type that is not part of the original JUEL source implementation. In some cases, some Java SE equivalent types have been added, usually with just subsets of the necessary methods because substituting or removing the unsupported type was not feasible or desirable.

- Mobile: added behavior

Used anywhere an area of functionality has been added to the original JUEL implementation. Additions are driven by ADFmc requirements and can generally be used or disabled without affecting the core behavior of the JUEL implementation.

- Mobile: changed behavior

Used anywhere an area of functionality has been altered from the original JUEL implementation. Reasons for this can be due to simplification, performance, reliance on unsupported types, or a specific need in an ADFmc use case.

- Mobile: removed behavior

Used anywhere code in the original JUEL implementation has been removed and not replaced. The feature requirements for the ADFmc EL implementation have been driven by the what is needed in initial ADFmc targeted apps. Functional areas that don't port over well, and don't have a pressing use case in ADFmc have been in many cases, removed (Again the original source is there and commented, or the removed type is still present in the 'unconverted' source folder).

- Mobile: renamed package

One family of devices targeted by ADFmc does not allow libraries or applications to define any classes as part of the packages `java.*` or `javax.*`. This necessitated a rename of the base EL interface package from `"javax.el"` to `"oracle.adfnmc.el"`.