OGNL Developer Guide

Drew Davidson

OGNL Developer Guide

Drew Davidson

Table of Contents

List of Examples

Chapter 1. Introduction

OGNL as a language allows for the navigation of Java objects through a concise syntax that allows for specifying, where possible, symmetrically settable and gettable values. The language specifies a syntax that attempts to provide as high a level of abstraction as possible for navigating object graphs; this usually means specifying paths through and to JavaBeans properties, collection indices, etc. rather than directly accessing property getters and setters (collectively know as *accessors*).

The normal usage of OGNL is to embed the language inside of other constructs to provide a place for flexible binding of values from one place to another. An example of this is a web application where values need to be bound from a model of some sort to data transfer objects that are operated on by a view. Another example is an XML configuration file wherein values are generated via expressions which are then bound to configured objects.

Embedding OGNL

The `ognl.Ognl` class contains convenience methods for evaluating OGNL expressions. You can do this in two stages, parsing an expression into an internal form and then using that internal form to either set or get the value of a property; or you can do it in a single stage, and get or set a property using the String form of the expression directly. It is more efficient to pre-compile the expression to it's parsed form, however, and this is the recommended usage.

OGNL expressions can be evaluated without any external context, or they can be provided with an execution environment that sets up custom extensions to modify the way that expressions are evaluated.

The following example illustrates how to encapsulate the parsing of an OGNL expression within an object so that execution will be more efficient. The class then takes an `OgnlContext` and a root object to evaluate against.

Example 1.1. Expression Class

```
import ognl.Ognl;
import ognl.OgnlContext;

public class OgnlExpression
{
    private Object       expression;

    public OgnlExpression(String expressionString) throws OgnlException
    {
        super();
        expression = Ognl.parseExpression(expressionString);
    }

    public Object getExpression()
    {
        return expression;
    }

    public Object getValue(OgnlContext context, Object rootObject) throws OgnlException
    {
        return Ognl.getValue(getExpression(), context, rootObject);
    }

    public void setValue(OgnlContext context, Object rootObject, Object value) throws OgnlException
    {
        Ognl.setValue(getExpression(), context, rootObject, value);
    }
}
```

Extending OGNL

OGNL expressions are not evaluated in a static environment, as Java programs are. Expressions are not compiled to bytecode at the expression level based on static class reachability. The same expression can have multiple paths through an object graph depending upon the root object specified and the dynamic results of the navigation. Objects that are delegated to handle all of the access to properties of objects,

elements of collections, methods of objects, resolution of class names to classes and converting between types are collectively known as *OGNL extensions*. The following chapters delve more deeply into these extensions and provide a roadmap as to how they are used within OGNL to customize the dynamic runtime environment to suit the needs of the embedding program.

Chapter 2. Property Accessors

When navigating an OGNL expression many of the elements that are found are properties. Properties can be many things depending on the object being accessed. Most of the time these property names resolve to JavaBeans properties that conform to the set/get pattern. Other objects (such as `Map`) access properties as keyed values. Regardless of access methodology the OGNL syntax remains the same. Under the hood, however, there are `PropertyAccessor` objects that handle the conversion of property name to an actual access to an objects' properties.

```
public interface PropertyAccessor
{
    Object getProperty( Map context, Object target, Object name ) throws OgnlException;
    void setProperty( Map context, Object target, Object name, Object value ) throws OgnlException;
}
```

You can set a property accessor on a class-by-class basis using `OgnlRuntime.setPropertyAccessor()`. There are default property accessors for `Object` (which uses JavaBeans patterns to extract properties) and `Map` (which uses the property name as a key).

Chapter 3. Method Accessors

Method calls are another area where OGNL needs to do lookups for methods based on dynamic information. The MethodAccessor interface provides a hook into how OGNL calls a method. When a static or instance method is requested the implementor of this interface is called to actually execute the method.

```
public interface MethodAccessor
{
    Object callStaticMethod( Map context, Class targetClass, String methodName, List args ) throws MethodFailedException;
    Object callMethod( Map context, Object target, String methodName, List args ) throws MethodFailedException;
}
```

You can set a method accessor on a class-by-class basis using `OgnlRuntime.setMethodAccessor()`. The is a default method accessor for `Object` (which simply finds an appropriate method based on method name and argument types and uses reflection to call the method).

Chapter 4. Elements Accessors

Since iteration is a built-in function of OGNL and many objects support the idea of iterating over the contents of an object (i.e. the `object.{ ... }` syntax) OGNL provides a hook into how iteration is done. The `ElementsAccessor` interface defines how iteration is done based on a source object. Simple examples could be a `Collection` elements accessor, which would simply

```
public interface ElementsAccessor
{
    public Enumeration getElements( Object target ) throws OgnlException;
}
```

You can set a method accessor on a class-by-class basis using `OgnlRuntime.setElementsAccessor()`. There are default elements accessors for `Object` (which returns an `Enumeration` of itself as the only object), `Map` (which iterates over the values in the `Map`), and Collection (which uses the collection's `iterator()`). One clever use of `ElementsAccessor`s is the `NumberElementsAccessor` class which allows for generating numeric sequences from 0 to the target value. For example the expression `(100).{ #this }` will generate a list of 100 integers ranged 0..99.

Chapter 5. Class References

In the sections on accessing static field and static methods it stated that classes must be full-specified in between the class reference specifier (@<classname>@<field|method>@). This is not entirely true; the default `ClassResolver` simply looks up the name of the class and assumes that it is fully specified. The `ClassResolver` interface is included in the OGNL context to perform lookup of classes when an expression is evaluated. This makes it possible to specify, for example, a list of imports that are specific to a particular `setValue()` or `getValue()` context in order to look up classes. It also makes class references agreeably short because you don't have to full specify a class name.

```
public interface ClassResolver
{
    public Class classForName(Map context, String className) throws ClassNotFoundException;
}
```

You can set a class resolver on a context basis using the `Ognl` methods `addDefaultContext()` and `createDefaultContext()`.

Chapter 6. Type Conversion

When performing set operations on properties or calling methods it is often the case that the values you want to set have a different type from the expected type of the class. OGNL supports a context variable (set by `OgnlRuntime.setTypeConverter(Map context, TypeConverter typeConverter)`) that will allow types to be converted from one to another. The default type converter that is uses is the `ognl.DefaultTypeConverter`, which will convert among numeric types (`Integer`, `Long`, `Short`, `Double`, `Float`, `BigInteger`, `BigDecimal`, and their primitive equivalents), string types (`String`, `Character`) and `Boolean`. Should you need specialized type conversion (one popular example is in Servlets where you have a `String[]` from an `HttpServletRequest.getParameters()` and you want to set values with it in other objects; a custom type converter can be written (most likely subclassing `ognl.DefaultTypeConverter`) to convert `String[]` to whatever is necessary.

```
public interface TypeConverter
{
    public Object convertValue(Map context,
                               Object target,
                               Member member,
                               String propertyName,
                               Object value,
                               Class toType);
}
```

Note that `ognl.DefaultTypeConverter` is much easier to subclass; it implements `TypeConverter` and calls it's own `convertValue(Map context, Object value, Class toType)` method and already provides the numeric conversions. For example, the above converter (i.e. converting `String[]` to `int[]` for a list of identifier parameters in a request) implemented as a subclass of `ognl.DefaultTypeConverter`:

```
HttpServletRequest request;
Map context = Ognl.createDefaultContext(this);

/* Create an anonymous inner class to handle special conversion */
Ognl.setTypeConverter(context, new ognl.DefaultTypeConverter() {
    public Object convertValue(Map context, Object value, Class toType)
    {
        Object  result = null;

        if ((toType == int[].class) && (value instanceof String[].class)) {
            String  sa = (String[])value;
            int[]   ia = new int[sa.length];

            for (int i = 0; i < sa.length; i++) {
                Integer     cv;

                cv = (Integer)super.convertValue(context,
                                                 sa[i],
                                                 Integer.class);
                ia[i] = cv.intValue();
            }
            result = ia;
        } else {
            result = super.convertValue(context, value, toType);
        }
        return result;
    }
});
/* Setting values within this OGNL context will use the above-defined TypeConverter */
Ognl.setValue("identifiers",
                context,
                this,
                request.getParameterValues("identifier"));
```

Chapter 7. Member Access

Normally in Java the only members of a class (fields, methods) that can be accessed are the ones defined with public access. OGNL includes an interface that you can set globally (using `OgnlContext.setMemberAccessManager()`) that allows you to modify the runtime in Java 2 to allow access to private, protected and package protected fields and methods. Included in the OGNL package is the `DefaultMemberAccess` class. It contains a constructor that allows you to selectively lower the protection on any private, protected or package protected members `using the AccessibleObject` interface in Java2. The default class can be subclasses to select different objects for which accessibility is allowed.

```
public interface MemberAccess
{
    public Object setup( Member member );
    public void restore( Member member, Object state );
    public boolean isAccessible( Member member );
}
```

Chapter 8. Null Handler

When navigating a chain sometimes properties or methods will evaluate to null, causing subsequent properties or method calls to fail with `NullPointerException`s. Most of the time this behaviour is correct (as it is with Java), but sometimes you want to be able to dynamically substitute another object in place of `null`. The `NullHandler` interface allows you to specify on a class-by-class basis how nulls are handled within OGNL expressions. Implementing this interface allows you to intercept when methods return `null` and properties evaluate to `null` and allow you to substitute a new value. Since you are given the source of the method or property a really clever implementor might write the property back to the object so that subsequent invocations do not return or evaluate to `null`.

```
public interface NullHandler
{
    public Object nullMethodResult(Map context, Object target, String methodName, List args);
    public Object nullPropertyValue(Map context, Object target, Object property);
}
```

`NullHandler` implementors are registered with OGNL using the `OgnlRuntime.setNullHandler()` method.

Chapter 9. Other API features

Tracing Evaluations

As of OGNL 2.5.0 the `OgnlContext` object can automatically tracks evaluations of expressions. This tracking is kept in the `OgnlContext` as currentEvaluation during the evaluation. After execution you can access the last evaluation through the lastEvaluation property of `OgnlContext`.

Note

The tracing feature is turned off by default. If you wish to turn it on there is a `setTraceEvaluations()` method on `OgnlContext` that you can call.

Any method accessor, elements accessor, type converter, property accessor or null handler may find this useful to give context to the operation being performed. The `Evaluation` object is itself a tree and can be traversed up, down and left and right through siblings to determine the exact circumstances of an evaluation. In addition the `Evaluation` object tracks the node that was performing the operation, the source object on which that operation was being performed and the result of the operation. If an exception is thrown during execution the user can get the last evaluation's last descendent to find out exactly which subexpression caused the error. The execption is also tracked in the `Evaluation`.