OGNL Language Guide

Drew Davidson

OGNL Language Guide

Drew Davidson

Table of Contents

List of Tables

Chapter 1. Introduction

OGNL stands for **O**bject **G**raph **N**avigation **L**anguage. It is an expression and binding language for getting and setting properties of Java objects. Normally the same expression is used for both getting and setting the value of a property.

We pronounce OGNL as a word, like the last syllables of a drunken pronunciation of "orthogonal."

Many people have asked exactly what OGNL is good for. Several of the uses to which OGNL has been applied are:

- A binding language between GUI elements (textfield, combobox, etc.) to model objects. Transformations are made easier by OGNL's TypeConverter mechanism to convert values from one type to another (String to numeric types, for example).
- A data source language to map between table columns and a Swing TableModel.
- A binding language between web components and the underlying model objects (WebOGNL, Tapestry, WebWork, WebObjects).
- A more expressive replacement for the property-getting language used by the Jakarata Commons BeanUtils package or JSTL's EL (which only allow simple property navigation and rudimentary indexed properties).

Most of what you can do in Java is possible in OGNL, plus other extras such as list *projection* and *selection* and *lambda expressions*.

Chapter 2. History

OGNL started out as a way to set up associations between UI components and controllers using property names. As the desire for more complicated associations grew, Drew Davidson created what he called KVCL, for Key-Value Coding Language, egged on by Luke Blanshard. Luke then reimplemented the language using ANTLR, came up with the new name, and, egged on by Drew, filled it out to its current state. Later on Luke again reimplemented the language using JavaCC. Further maintenance on all the code is done by Drew (with spiritual guidance from Luke).

Chapter 3. Syntax

Simple OGNL expressions are very simple. The language has become quite rich with features, but you don't generally need to worry about the more complicated parts of the language: the simple cases have remained that way. For example, to get at the name property of an object, the OGNL expression is simply name. To get at the text property of the object returned by the headline property, the OGNL expression is headline.text.

What is a property? Roughly, an OGNL property is the same as a bean property, which means that a pair of get/set methods, or alternatively a field, defines a property (the full story is a bit more complicated, since properties differ for different kinds of objects; see below for a full explanation).

The fundamental unit of an OGNL expression is the navigation chain, usually just called "chain." The simplest chains consist of the following parts:

Table 3.1. OGNL Expression Parts

Expression Element Part	Example
Property names	like the name and headline.text examples above
Method Calls	hashCode() to return the current object's hash code
Array Indices	listeners[0] to return the first of the current object's list of listeners

All OGNL expressions are evaluated in the context of a current object, and a chain simply uses the result of the previous link in the chain as the current object for the next one. You can extend a chain as long as you like. For example, this chain:

```
name.toCharArray()[0].numericValue.toString()
```

This expression follows these steps to evaluate:

- extracts the name property of the initial, or root, object (which the user provides to OGNL through the *OGNL context*)
- calls the toCharArray() method on the resulting `String`
- extracts the first character (the one at index 0) from the resulting array
- gets the numericValue property from that character (the character is represented as a `Character` object, and the `Character` class has a method called getNumericValue()).
- calls toString() on the resulting `Integer` object. The final result of this expression is the `String` returned by the last toString() call.

Note that this example can only be used to get a value from an object, not to set a value. Passing the above expression to the Ognl.setValue() method would cause an `InappropriateExpressionException` to be thrown, because the last link in the chain is neither a property name nor an array index.

This is enough syntax to do the vast majority of what you ever need to do.

Chapter 4. Expressions

This section outlines the details the elements of OGNL's expressions.

Constants

OGNL has the following kinds of constants:

- String literals, as in Java (with the addition of single quotes): delimited by single- or double-quotes, with the full set of character escapes.
- Character literals, also as in Java: delimited by single-quotes, also with the full set of escapes.
- Numeric literals, with a few more kinds than Java. In addition to Java's ints, longs, floats and doubles, OGNL lets you specify BigDecimals with a "b" or "B" suffix, and BigIntegers with an "h" or "H" suffix (think "huge"---we chose "h" for BigIntegers because it does not interfere with hexadecimal digits).
- Boolean (`true` and `false`) literals.
- The `null` literal.

Referring to Properties

OGNL treats different kinds of objects differently in its handling of property references. Maps treat all property references as element lookups or storage, with the property name as the key. Lists and arrays treat numeric properties similarly, with the property name as the index, but string properties the same way ordinary objects do. Ordinary objects (that is, all other kinds) only can handle string properties and do so by using "get" and "set" methods (or "is" and "set"), if the object has them, or a field with the given name otherwise.

Note the new terminology here. Property "names" can be of any type, not just Strings. But to refer to non-String properties, you must use what we have been calling the "index" notation. For example, to get the length of an array, you can use this expression:

```
array.length
```

But to get at element 0 of the array, you must use an expression like this:

```
array[0]
```

Note that Java collections have some special properties associated with them. See the section called “Pseudo-Properties for Collections”for these properties.

Indexing

As discussed above, the "indexing" notation is actually just property reference, though a computed form of property reference rather than a constant one.

For example, OGNL internally treats the "array.length" expression exactly the same as this expression:

```
array["length"]
```

And this expression would have the same result (though not the same internal form):

```
array["len" + "gth"]
```

Array and List Indexing

For Java arrays and Lists indexing is fairly simple, just like in Java. An integer index is given and that element is the referrent. If the index is out of bounds of the array or List and IndexOutOfBoundsException is thrown, just as in Java.

JavaBeans Indexed Properties

JavaBeans supports the concept of Indexed properties. Specifically this means that an object has a set of methods that follow the following pattern:

- public *`PropertyType`*[] get*`PropertyName`*()
- public void set*`PropertyName`*(*`PropertyType`*[] anArray)
- public *`PropertyType`* get*`PropertyName`*(int index)
- public void set*`PropertyName`*(int index, *`PropertyType`* value)

OGNL can interpret this and provide seamless access to the property through the indexing notation. References such as

```
someProperty[2]
```

are automatically routed through the correct indexed property accessor (in the above case through `getSomeProperty(2)` or `setSomeProperty(2, value)`). If there is no indexed property accessor a property is found with the name `someProperty` and the index is applied to that.

OGNL Object Indexed Properties

OGNL extends the concept of indexed properties to include indexing with arbitrary objects, not just integers as with JavaBeans Indexed Properties. When finding properties as candidates for object indexing, OGNL looks for patterns of methods with the following signature:

- public *`PropertyType`* get*`PropertyName`*(*`IndexType`* index)
- public void set*`PropertyName`*(*`IndexType`* index, *`PropertyType`* value)

The *`PropertyType`* and *`IndexType`* must match each other in the corresponding set and get methods. An actual example of using Object Indexed Properties is with the Servlet API: the Session object has two methods for getting and setting arbitrary attributes:

```
public Object getAttribute(String name) public void setAttribute(String name, Object value)
```

An OGNL expression that can both get and set one of these attributes is

```
session.attribute["foo"]
```

Calling Methods

OGNL calls methods a little differently from the way Java does, because OGNL is interpreted and must choose the right method at run time, with no extra type information aside from the actual arguments supplied. OGNL always chooses the most specific method it can find whose types match the supplied arguments; if there are two or more methods that are equally specific and match the given arguments, one of them will be chosen arbitrarily.

In particular, a null argument matches all non-primitive types, and so is most likely to result in an unexpected method being called.

Note that the arguments to a method are separated by commas, and so the comma operator cannot be used unless it is enclosed in parentheses. For example,

```
method( ensureLoaded(), name )
```

is a call to a 2-argument method, while

```
method( (ensureLoaded(), name) )
```

is a call to a 1-argument method.

Variable References

OGNL has a simple variable scheme, which lets you store intermediate results and use them again, or just name things to make an expression easier to understand. All variables in OGNL are global to the entire expression. You refer to a variable using a number sign in front of its name, like this:

```
#var
```

OGNL also stores the current object at every point in the evaluation of an expression in the this variable, where it can be referred to like any other variable. For example, the following expression operates on the number of listeners, returning twice the number if it is more than 100, or 20 more than the number otherwise:

```
listeners.size().(#this > 100? 2*#this : 20+#this)
```

OGNL can be invoked with a map that defines initial values for variables. The standard way of invoking OGNL defines the variables `root` (which holds the initial, or root, object), and `context` (which holds the `Map` of variables itself).

To assign a value to a variable explicitly, simply write an assignment statement with a variable reference on the left-hand side:

```
#var = 99
```

Parenthetical Expressions

As you would expect, an expression enclosed in parentheses is evaluated as a unit, separately from any surrounding operators. This can be used to force an evaluation order different from the one that would be implied by OGNL operator precedences. It is also the only way to use the comma operator in a method argument.

Chained Subexpressions

If you use a parenthetical expression after a dot, the object that is current at the dot is used as the current object throughout the parenthetical expression. For example,

```
headline.parent.(ensureLoaded(), name)
```

traverses through the headline and parent properties, ensures that the parent is loaded and then returns (or sets) the parent's name.

Top-level expressions can also be chained in this way. The result of the expression is the right-most expression element.

```
ensureLoaded(), name
```

This will call `ensureLoaded()` on the root object, then get the name property of the root object as the result of the expression.

Collection Construction

Lists

To create a list of objects, enclose a list of expressions in curly braces. As with method arguments, these expressions cannot use the comma operator unless it is enclosed in parentheses. Here is an example:

```
name in { null,"Untitled" }
```

This tests whether the `name` property is `null` or equal to `"Untitled"`.

The syntax described above will create a instanceof the `List` interface. The exact subclass is not defined.

Native Arrays

Sometimes you want to create Java native arrays, such as int[] or Integer[]. OGNL supports the creation of these similarly to the way that constructors are normally called, but allows initialization of the native array from either an existing list or a given size of the array.

```
new int[] { 1, 2, 3 }
```

This creates a new int array consisting of three integers 1, 2 and 3.

To create an array with all `null` or `0` elements, use the alternative size constructor

```
new int[5]
```

This creates an int array with 5 slots, all initialized to zero.

Maps

Maps can also be created using a special syntax.

```
#{ "foo" : "foo value", "bar" : "bar value" }
```

This creates a Map initialized with mappings for `"foo"` and `"bar"`.

Advanced users who wish to select the specific Map class can specify that class before the opening curly brace

```
#@java.util.LinkedHashMap@{ "foo" : "foo value", "bar" : "bar value" }
```

The above example will create an instance of the JDK 1.4 class `LinkedHashMap`, ensuring the the insertion order of the elements is preserved.

Projecting Across Collections

OGNL provides a simple way to call the same method or extract the same property from each element in a collection and store the results in a new collection. We call this "projection," from the database term for choosing a subset of columns from a table. For example, this expression:

```
listeners.{delegate}
```

returns a list of all the listeners' delegates. See the coercion section for how OGNL treats various kinds of objects as collections.

During a projection the `#this` variable refers to the current element of the iteration.

```
objects.{ #this instanceof String ? #this : #this.toString()}
```

The above would produce a new list of elements from the objects list as string values.

Selecting From Collections

OGNL provides a simple way to use an expression to choose some elements from a collection and save the results in a new collection. We call this "selection," from the database term for choosing a subset of rows from a table. For example, this expression:

```
listeners.{? #this instanceof ActionListener}
```

returns a list of all those listeners that are instances of the `ActionListener` class. See the coercion section for how OGNL treats various kinds of objects as collections.

Selecting First Match

In order to get the first match from a list of matches, you could use indexing such as `listeners.{? true }[0]`. However, this is cumbersome because if the match does not return any results (or if the result list is empty) you will get an `ArrayIndexOutOfBoundsException`.

The selection syntax is also available to select only the first match and return it as a list. If the match does not succeed for any elements an empty list is the result.

```
objects.{^ #this instanceof String }
```

Will return the first element contained in objects that is an instance of the `String` class.

Selecting Last Match

Similar to getting the first match, sometimes you want to get the last element that matched.

```
objects.{$ #this instanceof String }
```

This will return the last element contained in objects that is an instanceof the `String` class

Calling Constructors

You can create new objects as in Java, with the `new` operator. One difference is that you must specify the fully qualified class name for classes other than those in the java.lang package. [1] (for example, `new java.util.ArrayList()`, rather than simply `new ArrayList()`).

OGNL chooses the right constructor to call using the same procedure it uses for overloaded method calls.

Calling Static Methods

You can call a static method using the syntax `@class@method(args)`. If you leave out class, it defaults to `java.lang.Math`, to make it easier to call `min` and `max` methods. If you specify the class, you must give the fully qualified name.

If you have an instance of a class whose static method you wish to call, you can call the method through the object as if it was an instance method.

[1] This is only true with the default ClassResolver in place. With a custom class resolver packages can be mapped in such a way that more Java-like references to classes can be made. Refer to the OGNL Developer's Guide for details on using `ClassResolver` class.

If the method name is overloaded, OGNL chooses the right static method to call using the same procedure it uses for overloaded instance methods.

Getting Static Fields

You can refer to a static field using the syntax `@class@field`. The class must be fully qualified.

Expression Evaluation

If you follow an OGNL expression with a parenthesized expression, without a dot in front of the parentheses, OGNL will try to treat the result of the first expression as another expression to evaluate, and will use the result of the parenthesized expression as the root object for that evaluation. The result of the first expression may be any object; if it is an AST, OGNL assumes it is the parsed form of an expression and simply interprets it; otherwise, OGNL takes the string value of the object and parses that string to get the AST to interpret.

For example, this expression

```
#fact(30H)
```

looks up the fact variable, and interprets the value of that variable as an OGNL expression using the `BigInteger` representation of `30` as the root object. See below for an example of setting the `fact` variable with an expression that returns the factorial of its argument. Note that there is an ambiguity in OGNL's syntax between this double evaluation operator and a method call. OGNL resolves this ambiguity by calling anything that looks like a method call, a method call. For example, if the current object had a fact property that held an OGNL factorial expression, you could not use this approach to call it

```
fact(30H)
```

because OGNL would interpret this as a call to the fact method. You could force the interpretation you want by surrounding the property reference by parentheses:

```
(fact)(30H)
```

Pseudo-Lambda Expressions

OGNL has a simplified lambda-expression syntax, which lets you write simple functions. It is not a full-blown lambda calculus, because there are no closures---all variables in OGNL have global scope and extent.

For example, here is an OGNL expression that declares a recursive factorial function, and then calls it:

```
#fact = :[#this<=1? 1 : #this*#fact(#this-1)], #fact(30H)
```

The lambda expression is everything inside the brackets. The #this variable holds the argument to the expression, which is initially `30H`, and is then one less for each successive call to the expression.

OGNL treats lambda expressions as constants. The value of a lambda expression is the *AST* that OGNL uses as the parsed form of the contained expression.

Pseudo-Properties for Collections

There are some special properties of collections that OGNL makes available. The reason for this is that the collections do not follow JavaBeans patterns for method naming; therefore the `size()`, `length()`,

etc. methods must be called instead of more intuitively referring to these as properties. OGNL corrects this by exposing certain pseudo-properties as if they were built-in.

Table 4.1. Special Collections Pseudo-Properties

Collection	Special Properties

`Collection` (inherited by `Map`, `List` & `Set`)	`size`
	The size of the collection
	
	`isEmpty`
	Evaluates to `true` if the collection is empty

List	`iterator` Evalutes to an `Iterator` over the `List`.

`Map`	`keys`
	Evalutes to a `Set` of all keys in the `Map`.
	
	`values`
	Evaluates to a `Collection` of all values in the `Map`.
	
	  **Note**
	These properties, plus `size` and `isEmpty`, are different than the indexed form of access for `Maps` (i.e. `someMap["size"]` gets the `"size"` key from the map, whereas `someMap.size` gets the size of the `Map`.

`Set`	`iterator` Evalutes to an `Iterator` over the `Set`.

`Iterator`	`next`
	Evalutes to the next object from the `Iterator`.
	
	`hasNext`
	Evaluates to `true` if there is a next object available from the `Iterator`.

`Enumeration`	`next`
	Evalutes to the next object from the `Enumeration`.
	
	`hasNext`
	Evaluates to `true` if there is a next object available from the `Enumeration`.
	
	`nextElement`
	Synonym for `next`.
	
	`hasMoreElements`
	Synonym for `hasNext`.

Operators that differ from Java's operators

For the most part, OGNL's operators are borrowed from Java and work similarly to Java's operators. See the OGNL Reference for a complete discussion. Here we describe OGNL operators that are not in Java, or that are different from Java.

- The comma (,) or sequence operator. This operator is borrowed from C. The comma is used to separate two independent expressions. The value of the second of these expressions is the value of the comma expression. Here is an example:

```
ensureLoaded(), name
```

When this expression is evaluated, the ensureLoaded method is called (presumably to make sure that all parts of the object are in memory), then the name property is retrieved (if getting the value) or replaced (if setting).

- List construction with curly braces ({}). You can create a list in-line by enclosing the values in curly braces, as in this example:

```
{ null, true, false }
```

- The `in` operator (and `not in`, its negation). This is a containment test, to see if a value is in a collection. For example,

```
name in {null,"Untitled"} || name
```

- See the OGNL reference for a full list of operations

Setting values versus getting values

As stated before, some values that are gettable are not also settable because of the nature of the expression. For example,

```
names[0].location
```

is a settable expression - the final component of the expression resolves to a settable property.

However, some expressions, such as

```
names[0].length + 1
```

are not settable because they do not resolve to a settable property in an object. It is simply a computed value. If you try to evaluate this expression using any of the `Ognl.setValue()` methods it will fail with an `InappropriateExpressionException`.

It is also possible to set variables using get expressions that include the '`=`' operator. This is useful when a get expression needs to set a variable as a side effect of execution.

Chapter 5. Coercing Objects to Types

Here we describe how OGNL interprets objects as various types. See below for how OGNL coerces objects to booleans, numbers, integers, and collections.

Interpreting Objects as Booleans

Any object can be used where a boolean is required. OGNL interprets objects as booleans like this:

- If the object is a `Boolean`, its value is extracted and returned
- If the object is a `Number`, its double-precision floating-point value is compared with zero; non-zero is treated as `true`, zero as `false`.
- If the object is a `Character`, its boolean value is `true` if and only if its char value is non-zero.
- Otherwise, its boolean value is `true` if and only if it is non-`null`.

Interpreting Objects as Numbers

Numerical operators try to treat their arguments as numbers. The basic primitive-type wrapper classes (Integer, Double, and so on, including Character and Boolean, which are treated as integers), and the "big" numeric classes from the java.math package (BigInteger and BigDecimal), are recognized as special numeric types. Given an object of some other class, OGNL tries to parse the object's string value as a number.

Numerical operators that take two arguments use the following algorithm to decide what type the result should be. The type of the actual result may be wider, if the result does not fit in the given type.

- If both arguments are of the same type, the result will be of the same type if possible.
- If either argument is not of a recognized numeric class, it will be treated as if it was a `Double` for the rest of this algorithm.
- If both arguments are approximations to real numbers (`Float`, `Double`, or `BigDecimal`), the result will be the wider type.
- If both arguments are integers (`Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, or `BigInteger`), the result will be the wider type.
- If one argument is a real type and the other an integer type, the result will be the real type if the integer is narrower than "int"; `BigDecimal` if the integer is `BigInteger`; or the wider of the real type and `Double` otherwise.

Interpreting Objects as Integers

Operators that work only on integers, like the bit-shifting operators, treat their arguments as numbers, except that `BigDecimal`s and `BigInteger`s are operated on as `BigInteger`s and all other kinds of numbers are operated on as Longs. For the `BigInteger` case, the result of these operators remains a `BigInteger`; for the `Long` case, the result is expressed as the same type of the arguments, if it fits, or as a `Long` otherwise.

Interpreting Objects as Collections

The projection and selection operators (`e1.{e2}` and `e1.{?e2}`), and the `in` operator, all treat one of their arguments as a collection and walk it. This is done differently depending on the class of the argument:

- Java arrays are walked from front to back
- Members of `java.util.Collection` are walked by walking their iterators
- Members of `java.util.Map` are walked by walking iterators over their values
- Members of `java.util.Iterator` and `java.util.Enumeration` are walked by iterating them
- Members of `java.lang.Number` are "walked" by returning integers less than the given number starting with zero
- All other objects are treated as singleton collections containing only themselves

Appendix A. OGNL Language Reference

This section has a fairly detailed treatment of OGNL's syntax and implementation. See below for a complete table of OGNL's operators, a section on how OGNL coerces objects to various types, and a detailed description of OGNL's basic expressions.

Operators

OGNL borrows most of Java's operators, and adds a few new ones. For the most part, OGNL's treatment of a given operator is the same as Java's, with the important caveat that OGNL is essentially a typeless language. What that means is that every value in OGNL is a Java object, and OGNL attempts to coerce from each object a meaning appropriate to the situation it is used in (see the section on coercion).

The following table lists OGNL operators in reverse precedence order. When more than one operator is listed in the same box, these operators have the same precedence and are evaluated in left-to-right order.

Table A.1. OGNL Operators

Operator	`getValue()` Notes	`setValue()` Notes
e1, *e2* Sequence operator	Both `e1` and `e2` are evaluated with the same source object, and the result of `e2` is returned.	`getValue` is called on `e1`, and then `setValue` is called on `e2`.
e1 = *e2* Assignment operator	`getValue` is called on `e2`, and then `setValue` is called on `e1` with the result of `e2` as the target object.	Cannot be the top-level expression for `setValue`.
e1 ? *e2* : *e3* Conditional operator	`getValue` is called on `e1` and the result is interpreted as a boolean. `getValue` is then called on either `e2` or `e3`, depending on whether the result of `e1` was `true` or `false` respectively, and the result is returned.	`getValue` is called on `e1`, and then `setValue` is called on either `e2` or `e3`.
e1 || *e2*, e1 `or` *e2* Logical or operator	`getValue` is called on `e1` and the result is interpreted as a boolean. If `true`, that result is returned; if `false`, `getValue` is called on `e2` and its value is returned.	`getValue` is called on `e1`; if `false`, `setValue` is called on `e2`. Note that `e1` being `true` prevents any further setting from taking place.
e1 && *e2*, *e1* `and` *e2* Logical and operator	`getValue` is called on `e1` and the result is interpreted as a boolean. If `false`, that result is returned; if true, `getValue` is called on e2 and its value is returned.	`getValue` is called on `e1`; if `true`, `setValue` is called on `e2`. Note that `e1` being `false` prevents any further setting from taking place.
e1 | *e2*, *e1* `bor` *e2* Bitwise or operator	`e1` and `e2` are interpreted as integers and the result is an integer.	Cannot be the top-level expression passed to `setValue`.

Operator	`getValue()` Notes	`setValue()` Notes
e1 ^ *e2*, *e1* `xor` *e2* Bitwise exclusive-or operator	`e1` and `e2` are interpreted as integers and the result is an integer.	Cannot be the top-level expression passed to `setValue`.
e1 & *e2*, *e1* `band` *e2* Bitwise and operator	`e1` and `e2` are interpreted as integers and the result is an integer.	Cannot be the top-level expression passed to `setValue`.
e1 == *e2*, *e1* `eq` *e2* Equality test *e1* != *e2*, *e1* `neq` *e2* Inequality test	Equality is tested for as follows. If either value is `null`, they are equal if and only if both are `null`. If they are the same object or the `equals()` method says they are equal, they are equal. If they are both `Number`s, they are equal if their values as double-precision floating point numbers are equal. Otherwise, they are not equal. These rules make numbers compare equal more readily than they would normally, if just using the equals method.	Cannot be the top-level expression passed to `setValue`.
e1 < *e2*, *e1* `lt` *e2* Less than comparison *e1* <= *e2*, *e1* `lte` *e2* Less than or equals comparison *e1* > *e2*, *e1* `gt` *e2* Greater than comparison *e1* >= *e2*, *e1* `gte` *e2* Greater than or equals comparison *e1* `in` *e2* List membership comparison *e1* `not in` *e2* List non-membership comparison	The ordering operators compare with `compareTo()` if their arguments are non-numeric and implement `Comparable`; otherwise, the arguments are interpreted as numbers and compared numerically. The in operator is not from Java; it tests for inclusion of e1 in e2, where e2 is interpreted as a collection. This test is not efficient: it iterates the collection. However, it uses the standard OGNL equality test.	Cannot be the top-level expression passed to `setValue`.
e1 << *e2*, *e1* `shl` *e2* Bit shift left *e1* >> *e2*, *e1* `shr` *e2* Bit shift right *e1* >>> *e2*, *e1* `ushr` *e2* Logical shift right	`e1` and `e2` are interpreted as integers and the result is an integer.	Cannot be the top-level expression passed to `setValue`.
e1 + *e2* Addition *e1* - *e2* Subtraction	The plus operator concatenates strings if its arguments are non-numeric; otherwise it interprets its arguments as numbers and adds	Cannot be the top-level expression passed to `setValue`.

Operator	`getValue()` Notes	`setValue()` Notes
	them. The minus operator always works on numbers.	
e1 * *e2* Multiplication *e1* / *e2* Division *e1* % *e2* Remainder	Multiplication, division, which interpret their arguments as numbers, and remainder, which interprets its arguments as integers.	Cannot be the top-level expression passed to `setValue`.
+ *e* Unary plus - *e* Unary minus ! *e*, not *e* Logical not ~ *e* Bitwise not *e* instanceof *class* Class membership	Unary plus is a no-op, it simply returns the value of its argument. Unary minus interprets its argument as a number. Logical not interprets its argument as a boolean. Bitwise not interprets its argument as an integer. The *class* argument to instanceof is the fully qualified name of a Java class.	Cannot be the top-level expression passed to `setValue`.
e.*method*(*args*) Method call *e*.*property* Property *e1*[*e2*] Index *e1*.{ *e2* } Projection *e1*.{? *e2* } Selection *e1*.(*e2*) Subexpression evaluation *e1*(*e2*) Expression evaluation	Generally speaking, navigation chains are evaluated by evaluating the first expression, then evaluating the second one with the result of the first as the source object.	Some of these forms can be passed as top-level expressions to `setValue` and others cannot. Only those chains that end in property references (e.property), indexes (`e1[e2]`), and subexpressions (`e1.(e2)`) can be; and expression evaluations can be as well. For the chains, `getValue` is called on the left-hand expression (`e` or `e1`), and then `setValue` is called on the rest with the result as the target object.
constant Constant (*e*) Parenthesized expression *method*(*args*) Method call	Basic expressions	Only property references (`property`), indexes (`[e]`), and variable references (`#variable`) can be passed as top-level expressions to `setValue`. For indexes, `getValue` is called on `e`, and then the result is used as the property "name" (which might be a `String` or any other kind of ob-

Operator	`getValue()` Notes	`setValue()` Notes

property
Property reference
 	
`[` *e* `]`
Index reference
 	
`{` *e*`, ... }`
List creation
 	
`#`*variable*
Context variable reference
 	
`@`*class*`@`*method*`(`*args*`)`
Static method reference
 	
`@`*class*`@`*field*
Static field reference
 	
`new` *class*`(`*args*`)`
Constructor call
 	
`new` *array-component-class*`[] {` *e*`, ... }`
Array creation
 	
`#{` *e1* `:` *e2*`, ... }`
Map creation
 	
`#@`*classname*`@{` *e1* `:` *e2*`, ... }`
Map creation with specific subclass
 	
`:[` *e* `]`
Lambda expression definition
		ject) to set in the current target object. Variable and property references are set more directly.

 **Note** These operators are listed in reverse precedence order		