

Expressions Framework Reference

Sven Efftinge, sven@efftinge.de, www.efftinge.de

PRIMARY SPONSORS



SECONDARY SPONSOR



Table of Contents

INTRODUCTION.....	3
TYPE SYSTEM.....	3
METAMODEL.....	3
TYPES.....	4
<i>Type Names.....</i>	4
<i>Collection Type Names.....</i>	4
<i>Features.....</i>	5
BUILT-IN TYPES.....	5
<i>Object.....</i>	5
<i>Void.....</i>	5
<i>Simple types (Datatypes).....</i>	5
<i>Collection types.....</i>	6
<i>Type types.....</i>	6
EXPRESSIONS SUB-LANGUAGE.....	7
LITERALS AND SPECIAL OPERATORS FOR BUILT-IN TYPES.....	7
<i>Object.....</i>	7
<i>Void.....</i>	7
<i>Type literals.....</i>	7
<i>StaticProperty literals.....</i>	7
<i>String.....</i>	8
<i>Boolean.....</i>	8
<i>Integer and Real.....</i>	8
<i>Collections.....</i>	9
SPECIAL COLLECTION OPERATIONS.....	9
<i>Select.....</i>	9
<i>TypeSelect.....</i>	9
<i>Reject.....</i>	9
<i>Collect.....</i>	10
<i>Shorthand for Collect (and more than that).....</i>	10
<i>forAll.....</i>	10
<i>Exists.....</i>	11
IF EXPRESSION.....	11
SWITCH EXPRESSION.....	11
CHAIN EXPRESSION.....	11
CREATE EXPRESSION.....	12
LET EXPRESSION.....	12

MULTI METHODS (MULTIPLE DISPATCH)..... 12
CASTING..... 13

Introduction

The oAW4 generator framework provides different textual languages, that are useful in different contexts in the MDSD process (e.g. checks, extensions, code generation, model transformation). The different languages have one thing in common: they operate on models, that are based on different metamodels and meta-metamodels.

The expressions framework provides a uniform abstraction layer over different meta-meta models (e.g. EMF's Ecore, JMI's MOF, JavaBeans, etc.). Additionally, it offers a expressions sub language, which is reused in the various textual languages.

Type System

The abstraction layer on API basis is called a *type system*. It provides access to built-in types and different registered metamodel implementations. These registered metamodel implementations offer access to the types they provide. The first part of this documentation describes the type system. The expression sub-language is described afterwards in the second part of this documentation. This differentiation is necessary because the type system and the expression language are two different things. The type system is a kind of reflection layer, that can be extended with metamodel implementations. The expression language defines a concrete syntax for executable expressions, that uses the type system.

The Java API described here is located in the *org.openarchitectureware.type* package and is a part of the subproject *core-expressions*

MetaModel

By default the type system only knows the built-in types. In order to register your own meta types, you need to register an implementation of *MetaModel* with the type system. This is usually done by configuring a property in the workflow component description. (See *workflow* reference for details)

A possible configuration of the *Xpand2* generator component looks like this:

```
<component class="org.openarchitectureware.xpand2.Generator">
  <metaModel class="org.openarchitectureware.type.emf.EmfMetaModel">
    <metaModelPackage value="org.eclipse.emf.ecore.EcorePackage"/>
  </metaModel>
  ...
</component>
```

In this example the *EmfMetaModel* implementation is configured. This means that it will be registered with the type system. The *metaModelPackage* property is a property that is specific to the *EmfMetaModel* (located in the *core-emftools* project).

A types, a specific meta model implementation provides and manages, can be retrieved by a query. You can either query by name or by `java.lang.Object`. The following is the `MetaModel` interface:

```
package org.openarchitectureware.type;

public interface MetaModel {
    TypeSystem getTypeSystem();
    void setTypeSystem(TypeSystem typeSystem);
    Type getTypeForName(String typeName);
    Type getType(Object obj);
    Set getKnownTypes();
    String getName();
}
```

Types

Every object (e.g. model elements, values, etc.) has a type. A type contains properties and operations. In addition it might inherit from other types (multiple inheritance).

Type Names

Types have a simple Name (e.g. `String`) and an optional namespace used to distinguish between two types with the same name (e.g. `my::metamodel`). The delimiter for name space fragments is a double colon `::`. A fully qualified name looks like this:

```
my::fully::qualified::MetaType
```

The namespace and name used by a specific type is defined by the corresponding `MetaModel` implementation. The `EmfMetaModel`, for instance, maps `EPackages` to namespace and `EClassifiers` to names. Therefore, the name of the `Ecore` element `EClassifier` is called:

```
ecore::EClassifier
```

If you don't want to use namespaces (for whatever reason), you can always implement your own metamodel and map the names accordingly.

Collection Type Names

The built-in type system also contains the following collection types: `Collection`, `List` and `Set`. Because the expressions language is statically type checked and we don't like casts and `ClassCastException`s, we introduced the concept of parameterized types. The typesystem doesn't support full featured generics, because we don't need them.

The syntax is:

```
Collection[my::Type]
List[my::Type]
Set[my::Type]
```

Features

Each type offers features. The type (resp. the metamodel) is responsible for mapping the features. There are three different kinds of features:

- Properties
- Operations
- StaticProperties

Properties are straight forward: They have a name and a type. They can be invoked on instances of the corresponding type. The same is true for *Operations*. But in contrast to properties, they can have parameters. *StaticProperties* are the equivalent to enums. They must be invoked statically and they don't have parameters.

Built-In types

As mentioned before the expressions framework has several built-in types, that define operations and properties. In the following, we'll give a rough overview of the types and their features. We won't document all of the operations here, because the built-in types will evolve over time and we want to derive the documentation from the implementation (model-driven :-)). For a complete reference consult the generated API documentation (<http://www.openarchitectureware.org/api/built-ins/>).

Object

Object defines a couple of basic operations, like equations. Every type has to extend *Object!*

Void

The void type can be specified as the return type for operations, although it's not recommended, because whenever possible expressions should be free of side effects whenever possible.

Simple types (Datatypes)

The type system doesn't have a concept data type. Data types are just types. As in OCL, we support the following types: *String*, *Boolean*, *Integer*, *Real*.

- **String:** A rich and convenient `String` library is especially important for code generation. The type system supports the '+' operator for concatenation, the usual `java.lang.String` operations (`length()`, etc.) and some special operations (like `firstUpper()`, `firstLower()`, often needed in code generation templates).
- **Boolean:** Boolean offers the usual operators (Java syntax): `&&`, `||`, `!`, etc.

- **Integer and Real:** Integer and Real offer the usual compare operators (<,>,<=,>=) and simple arithmetics (+,-,*,/). Note that *Integer extends Real*

Collection types

The type system has three different Collection types. *Collection* is the base type, it provides several operations known from `java.util.Collection`. The other two types (*List*, *Set*) correspond to their `java.util` equivalents, too.

Type types

The type system describes itself, hence, there are types for the different concepts. These types are needed for reflective programming. To avoid confusion with meta types with the same name (it's not unusual to have a meta type called *Operation*, for instance) we have prefixed all of the types with the namespace *oaw*. We have:

- *oaw::Type*
- *oaw::Feature*
- *oaw::Property*
- *oaw::StaticProperty*
- *oaw::Operation*

Expressions sub-language

The oAW expression sub language is a syntactical mixture of Java and OCL. This documentation provides a detailed description of each available expression. But let's start with some simple examples.

Accessing a property:

```
myModelElement.name
```

Accessing an operation:

```
myModelElement.doStuff()
```

simple arithmetic:

```
1 + 1 * 2
```

boolean expressions (just an example:-):

```
!('text'.startsWith('t')) && ! false
```

Literals and special operators for built-in types

There are several literals for built-in types:

Object

There are no literals for object, but we have two operators:

equals:

```
obj == obj
```

not equals:

```
obj != obj
```

Void

The only possible instance of Void is the null reference. Therefore, we have one literal:

```
null
```

Type literals

The literal for types is just the name of the type (no '.class' suffix, etc.). Example:

```
String // the type string  
my::special::Type // evaluates to the type 'my::special::Type'
```

StaticProperty literals

The literal for static properties (aka *enum* literals) is correlative to type literals:

```
my::Color::RED
```

String

There are two different literal syntaxes (with the same semantics):

```
'a String literal'  
"a String literal" // both are okay
```

For Strings the expression sub-language supports the plus operator that is overloaded with concatenation:

```
'my element '+ ele.name +' is really cool!'
```

Boolean

The boolean literals are:

```
true  
false
```

Operators are:

```
true && false // AND  
true || false // OR  
! true       // NOT
```

Integer and Real

The syntax for integer literals is as expected:

```
// integer literals  
3  
57278  
  
// real literals  
3.0  
0.75
```

Additionally, we have the common arithmetic operators:

```
3 + 4 // addition  
4 - 5 // subtraction  
2 * 6 // multiplication  
3 / 64 // divide  
  
// Unary minus operator  
- 42  
- 47.11
```

Furthermore, the well known compare operators are defined:

```
4 > 5 // greater than  
4 < 5 // smaller than  
4 >= 23 // greater equals than  
4 <= 12 // smaller equals than
```

Collections

There is a literal for lists:

```
{1,2,3,4} // a list with four integers
```

There is no other special concrete syntax for collections. If you need a set, you have to call the `asSet()` operation on the list literal:

```
{1,2,4,4}.asSet() // a set with 3(!) integers
```

Special Collection operations

Like OCL, the oAW expression sub-language defines several special operations on collections. Those operations are not members of the type system, therefore you can't use them in a reflective manner!

Select

Sometimes an expression yields a large collection, but one is only interested in a special subset of the collection. The expression sub-language has special constructs to specify a selection out of a specific collection. These are the `select` and `reject` operations. The `select` specifies a subset of a collection. A `select` is an operation on a collection and is specified as follows:

```
collection.select( v | boolean-expression-with-v )
```

`select` returns a sublist of the specified collection. The list contains all elements for which the evaluation of `boolean-expression-with-v` results in `true`. Example:

```
{1,2,3,4}.select(i|i>=3) // returns {3,4}
```

TypeSelect

A special version of a `select` expression is `typeSelect`. Rather than providing a boolean expression a class name is here provided.

```
collection.typeSelect( classname )
```

`TypeSelect` returns that sublist of the specified collection, that contains only objects which are an instance of the specified class (also inherited).

Reject

The `reject` operation is similar to the `select` operation, but with `reject` we get the subset of all the elements of the collection for which the expression evaluates to `False`. The `reject` syntax is identical to the `select` syntax:

```
collection.reject( v | boolean-expression-with-v )
```

Example:

```
{1,2,3,4}.reject(i|i>=3) // returns {1,2}
```

Collect

As shown in the previous section, the select and reject operations always result in a sub-collection of the original collection. Sometimes one wants to specify a collection which is derived from another collection, but which contains objects not in the original collection (it is not a sub-collection), we can use a `collect` operation. The collect operation uses the same syntax as the select and reject and is written like this:

```
collection.collect( v | expression-with-v )
```

Collect again iterates over the target collection and evaluates the given expression on each element. In contrast to select, the evaluation result is collected in a list. When an iteration is finished the list with all results is returned. Example:

```
namedElements.collect(ne|ne.name) // returns a list of strings
```

Shorthand for Collect (and more than that)

As navigation through many objects is very common, there is a shorthand notation for collect that makes the expressions more readable. Instead of

```
self.employee.collect(e|e.birthdate)
```

one can also write:

```
self.employee.birthdate
```

In general, when a property is applied to a collection of Objects, it will automatically be interpreted as a `collect` over the members of the collection with the specified property.

The syntax is a shorthand for collect, if the feature does not return a collection itself. But sometimes we have the following:

```
self.buildings.rooms.windows // returns a list of windows
```

This syntax works, but one cannot express it using the `collect` operation in an easy way.

forAll

Often a boolean expression has to be evaluated for all elements in a collection. The `forAll` operation allows specifying a `Boolean` expression, which must be true for all objects in a collection in order for the `forAll` operation to return true:

```
collection.forAll( v | boolean-expression-with-v )
```

The result of `forAll` is true if `boolean-expression-with-v` is true for all the elements contained in a collection. If `boolean-expression-with-v` is false for one or more of the elements in the collection, then the `forAll` expression evaluates to false. Example:

```
{3,4,500}.forAll(i|i<10) // evaluates to false (500 < 5 is false)
```

Exists

Often you will need to know whether there is at least one element in a collection for which a boolean is true. The *exists* operation allows you to specify a *Boolean* expression which must be true for at least one object in a collection:

```
collection.exists( v | boolean-expression-with-v )
```

The result of the *exists* operation is *true* if *boolean-expression-with-v* is true for at least one element of *collection*. If the *boolean-expression-with-v* is *false* for all elements in *collection*, then the complete expression evaluates to false. Example:

```
{3,4,500}.exists(i|i<10) // evaluates to true (e.g. 3 < 5 is true)
```

If expression

There are two different „flavours“ of conditional expressions. The first one is the so-called *if expression*. Syntax:

```
condition ? thenExpression : elseExpression
```

Example:

```
name != null ? name : 'unknown'
```

Switch expression

The other one is called *switch expression*. Syntax:

```
switch (expression) {  
  (case expression : thenExpression)*  
  default : catchAllExpression  
}
```

The default part is mandatory, because *switch* is an expression, therefore it needs to evaluate to something in any case. Example:

```
switch (person.name) {  
  case 'Hansen' : 'Du kanns platt schnacken'  
  default : 'Du kanns mi nech verstohn!'  
}
```

There is an abbreviation for *Boolean* expressions:

```
switch {  
  case booleanExpression : thenExpression  
  default : catchAllExpression  
}
```

Chain expression

Expressions and functional languages should be free of side effects as far as possible. But sometimes there you need invocations that do have side effects. In some cases expressions

even don't have a return type (i.e. the return type `void`). If you need to call such operations, you can use the chain expression. Syntax:

```
anExpr ->  
anotherExpr ->  
lastExpr
```

Each expression is evaluated in sequence, but only the result of the last expression is returned. Example:

```
pers.setName('test') ->  
pers
```

This chain expression will set the *name* of the person first, before it returns the *person* object itself.

Create expression

The create expression is used to instantiate new objects of a given type:

```
new TypeName
```

Let expression

The let expression lets you define local variables. Syntax is as follows:

```
let v = expression in expression-with-v
```

This is especially useful together with a chain- and a create expressions. Example:

```
let p = new Person in  
  p.name('Mork vom Ork') ->  
  p.age(42) ->  
  p.city('New York') ->  
  p
```

Multi methods (multiple dispatch)

The expressions language supports multiple dispatching. This means that when there is a bunch of overloaded operations, the decision which operation has to be resolved is based on the dynamic type of all parameters (the implicit 'this' included).

In Java only the dynamic type of the 'this' element is considered, for parameters the static type is used. (this is called single dispatch)

Here is a Java example:

```
class MyClass {  
  boolean equals(Object o) {  
    if (o instanceof MyClass) {  
      return equals((MyClass)o);  
    }  
  }  
}
```

```
    }  
    return super.equals(o);  
...}  
    boolean equals(MyType mt) {  
        //implementation...  
    }  
}
```

The method `equals(Object o)` must not be overwritten, if Java would support multiple dispatch.

Casting

The expression language is statically type checked. Although there are many concepts that help the programmer to have really good static type information, sometimes one knows more about the real type than the system. To explicitly give the system such an information casts are available. **Casts are 100% static, so you don't need them if you never statically typecheck your expressions!**

The syntax for casts is very Java-like:

```
((String) unTypedList.get(0)).toUpperCase()
```

About our Sponsors

itemis GmbH & Co. KG is an independent IT service company with an emphasis on consulting, coaching, and software development. Every single itemis expert provides many years of project experience and widespread knowledge about all object oriented and component based software development issues - especially in the field of model driven software development.

b+m is the founder of the openArchitectureWare project. The software was originally developed within the scope of many successful projects. b+m opened the software to the community in late 2003. All of the paradigms of Model-Driven Software Development including Product Line Engineering and not only the generator framework have become a key concept for product and customer specific development at b+m. b+m customers can make use of long time experience and substantial know-how in that field. Located at the company headquarters in Melsdorf/Kiel and at its subsidiaries in Berlin, Cottbus, Hamburg, Hanover and Kiel the b+m staff of 205 provides practical solutions for customized business applications, business process optimization and comprehensive architecture, project and quality management.

oose Innovative Informatik GmbH offers coaching, consulting and training in all themes about software engineering. The main focus of their activities are software architecture, requirements engineering and project-management. oose have first-hand information and experience, because our staff take actively part with others in actual trends, standards and innovations. Our staff support this and pass their know-how regularly on by writing and publishing books or being speaker at conferences, etc. Within the OMG oose collaborate actively on the specifications of the UML and also the SysML.

MID Enterprise Software Solutions GmbH is a leading supplier of optimized tool environments for standardsbased and model-centric software development as well as business process modeling. This includes professional tool consulting and tool components to build a complete tool environment using the best techniques and tool modules available - Architectural and Operational Excellence. With innovatorAOX, MID provides a holistic standard tool environment for object- and function-oriented software development as well as business process and data modeling to help its customers establish highly efficient processes and tool environments for software production, The unique and seamless integration of business process modeling into the development process ensures an unprecedented level of convergence of business requirements and implemented IT systems. Project members from all departments speak the same language and all requirements are clearly described.