# Extend Language Reference

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

# Table of Contents

## Introduction

Like the *expressions* sub language that summarizes the syntax of expressions for all the other textual languages delivered with the openArchitectureWare framework, there is another commonly used language called *Extend*.

This language provides the possibility to define rich libraries of independent operations based on either Java methods or oAW expressions. Those libraries can be referenced from all other textual languages, that are based on the expressions framework.

## Extend files

An *extend* file must reside in the Java class path of the used execution context. Additionally it's file extension must be *ext*. Let's have a look at an extend file.

```
import my::metamodel;
extension other::ExtensionFile;

/**
 * Documentation
 /
anExpressionExtension(String stringParam) :
   doingStuff(with(stringParam))
;

/**
 * java extensions are just mappings
 */
String aJavaExtension(String param) :
   JAVA my.JavaClass.staticMethod(java.lang.String)
;
```

The example shows all available statements. The structure of such a file is:

- import statements
- extension import statements
- expression or java extensions

## Comments

We have single- and multi-line comments.

The syntax for single line comments is:

```
// my comment
```

Multi line comments are written like this:

```
/* My
   multi line
   comment */
```

## Import Statements

Using the import statement one can import name spaces of different types.(see expressions framework reference documentation).

Syntax is:

```
import my::imported::namespace;
```

Extend does not support static imports or any similar concept. Therefore, the following is incorrect syntax:

```
import my::imported::namespace::*; // WRONG!
import my::Type;                   // WRONG!
```

## Extension Import Statement

You can import another extend file using the extension statement. The syntax is:

```
extension fully::qualified::ExtensionFileName;
```

Note, that no file extension (*.ext) is specified.

## Extensions

Extensions are the main reason for this document. A syntax of an extension definition follows:

```
ReturnType extensionName(ParamType1 paramName1, ParamType2...):
   expression-using-params
;
```

Example:

```
String getterName(NamedElement ele) :
   'get'+ele.name.firstUpper()
;
```

### Extension Invocation

There are two different ways, of how to invoke an extension. It can be invoked like a function (Actually an extension is a function unless its return type is `Void`):

```
getterName(myNamedElement)
```

The other way to invoke an extension is through a „pseudo member syntax“:

```
myNamedElement.getterName()
```

For any invocation in member syntax, the target expression (the member) is mapped to the first parameter. Therefore, both syntax forms do the same thing.

It's important to understand, that extensions are not members of the type system, hence, they are not accessible through reflection and you cannot specialize or overwrite operations using them.

The expression evaluation engine first looks for an appropriate operation before looking for an extension, i.e. operations have higher precedence.

### Type Inference

For most extensions, you don't need to specify the return type, because it can be derived from the specified expression. The special thing is, that the static return type of such an extension depends on the context of use!

For instance, if you have the following extension

```
asList(Object o): {o};
```

the invocation of

```
asList('text')
```

has the static type `List[String]`. This means you can call

```
asList('text').get(0).toUpperCase()
```

The expression is statically type safe!

### Recursion

There is only one exception to the statement, that you don't have to specifiy the return type: recursive extensions.

```
String fullyQualifiedName(NamedElement n) :
   n.parent == null ? n : fullyQualifiedName(n)+'::'+n
;
```

Recursive extensions are non-deterministic in a static context, therefore it is necessary to specify a return type.

## Java Extensions

In some rare cases one does want to call a Java method from inside an expression. This can be done by providing a Java extension:

```
Void myJavaExtension(String param) :
   JAVA my.Type.staticMethod(java.lang.String)
;
```

The signature is the same as for any other extension. The implementation is redirected to a public static method in a Java class.

It's syntax is:

```
JAVA fully.qualified.Type.staticMethod(my.ParamType1,
                                       my.ParamType2,
                                       ...)
```

Note that you cannot use imported namespaces (we can import *oAW* namespaces, only). You have to specify the type, its method and the parameter types in a fully qualified way.

Example:

If you have defined the following Java extension:

```
Void dump(String s) :
 JAVA my.Helper.dump(java.lang.String)
;
```

and you have the following Java class:

```
package my;
public class Helper {
   public final static void dump(String aString) {
      System.out.println(aString);
   }
}
```

the expressions

```
dump('Hello world!')
'Hello World'.dump()
```

both result are invoking the Java method `void dump(String aString)`

## Calling Extensions From Java

The previous section showed how to implement Extensions in Java. This section shows you how to call non-Java Extensions from Java.

```
// setup
XtendFacade f = XtendFacade.create("my::path::MyExtensionFile");

// use
f.call("sayHello",new Object[]{"World"});
```

The called extension file looks like this:

```
appendStuff(String s) :
      "Hello " + s;
```

This example uses only features of the BuiltinMetaModel, in this case the „+“ feature from the StringTypeImpl.

Here is another example, that uses the JavaBeansMetaModel strategy. This strategy provides as additional feature the access to properties using the getter and setter methods.

For more information about type systems see the Expressions Reference Documentation.

We have one javaBean-like meta model class:

```
package mypackage;
public class MyBeanMetaClass {
	private String myProp;
	public String getMyProp() { return myProp; }
	public void setMyProp(String s) { myProp = s; }
}
```

Additional to the already builtin metamodel type system, we register the JavaMetaModel with the JavaBeansStrategy for our facade. Now we can use also this strategy in our extension:

```
// setup facade
XtendFacade f = XtendFacade.create("myext::JavaBeanExtension");

// setup additional type system
JavaMetaModel jmm =
	new JavaMetaModel("JavaMM", new JavaBeansStrategy());
f.registerMetaModel(jmm);

// use the facade
MyBeanMetaClass jb = MyBeanMetaClass();
jb.setMyProp("test");
f.call("readMyProp",new Object[]{jb}));
```

The called extension file looks like this:

```
import mypackage;

readMyProp(MyBeanMetaClass jb) :
	jb.myProp;
```

# 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 expirience 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.