

# Wombat Language Reference

*Arno Haase, [arno.haase@haase-consulting.com](mailto:arno.haase@haase-consulting.com)*

## Table of Contents

<b>INTRODUCTION.....</b>	<b>4</b>
<b>KEY CONCEPTS.....</b>	<b>4</b>
METAMODELS.....	4
MODELS.....	4
TRANSFORMATIONS AND MODIFICATIONS.....	5
<b>WOMBAT SCRIPTS.....</b>	<b>5</b>
SCRIPT FILES.....	5
TOP-LEVEL ELEMENTS.....	6
COMMENTS.....	6
WHITESPACE.....	6
IDENTIFIERS.....	6
<b>TYPES.....</b>	<b>6</b>
BUILT-IN TYPES.....	6
USER-SUPPLIED TYPES.....	7
<b>FUNCTIONS.....</b>	<b>7</b>
SYNTAX.....	7
CACHING OF RESULTS.....	8
CALLING A WOMBAT SCRIPT.....	8
LAZY EVALUATION.....	8
<b>EXPRESSIONS.....</b>	<b>9</b>
INITIALIZATION EXPRESSION.....	9
CONDITIONAL EXPRESSION.....	9
CHAIN EXPRESSION.....	9
LOGICAL “OR” EXPRESSION.....	10
LOGICAL “AND” EXPRESSION.....	10
COMPARISON EXPRESSIONS.....	10
ARITHMETIC EXPRESSIONS.....	10
IDENTIFIER EXPRESSIONS.....	11
PROPERTY EXPRESSIONS.....	11
LOGICAL AND BITWISE “NOT” EXPRESSION.....	11
EXTENSION CALL EXPRESSION.....	11
TYPE LITERAL EXPRESSION.....	12
PRIMITIVE LITERAL EXPRESSION.....	12
COLLECTION LITERALS.....	12

COLLECTION LOOPS.....	12
PARENTHESES.....	13
OBJECT CREATION.....	13
OBJECT MODIFICATION.....	13
“SWITCH”.....	14
COLLECTION ARITHMETIC.....	14
PREDICATES ON COLLECTIONS.....	14
<b>POLYMORPHIC INVOCATION.....</b>	<b>15</b>
<b>EXTENSIONS.....</b>	<b>16</b>
INTERNAL DEFINITION.....	16
EXTERNAL DEFINITION.....	16

## Introduction

Wombat is a lightweight engine for transforming and/or modifying models. This document describes both the Wombat language and the fundamental underlying APIs.

It takes one or more instantiated models as input and provides one or more instantiated models as output. These models can have different metamodels and even metametamodels (JavaBeans, EMF, classic, ...), making Wombat useful for merging arbitrary models or converting from one metametamodel to another.

Wombat deals only with instantiated models, i.e. object graphs in memory. Therefore it must always be used as part of a tool chain that does the instantiation and handles the resulting models.

## Key concepts

To start with, it is important the things Wombat works on and what it does to them. The concepts presented in this section will become more tangible later on in this documentation when concrete syntax is introduced.

It is however fundamental for understanding Wombat to have a clear picture of these concepts in mind up front, and hence this abstract explanation.

## Metamodels

Metamodels are explicit in Wombat. The key feature of a metamodel is that it can perform the mapping between a string representation and a type. Every type can belong to only one metamodel. Every metamodel has a name by which it is globally known.

An actual type can for be a JavaBeans compliant class or an EMF EClass for example, and each of these metametamodels comes with its own style of string representation for a type. It is the responsibility of the metamodel to interpret the string representation of a type. The same string representation can reference different types in different metamodels.

So in order to specify which type is meant, it is necessary to have both the name of the metamodel and string representation of the type relative to the metamodel.

## Models

In Wombat, every “real” object (as opposed to strings etc.) belongs to exactly one *model*, and this relationship is explicit. Every model has a name, and every model has exactly one corresponding metamodel.

Models know which elements belong to them. And in order to query all elements of a given type, it is necessary to specify a model: Models contain elements, types by themselves have no knowledge of their instances.

So whenever a new object is instantiated in Wombat, it is necessary to specify to which model it should be added.

## Transformations and Modifications

Wombat was designed to be excellent at model *transformations*, i.e. creating new models based on the original models but leaving the original models unchanged. That approach simplifies writing the scripts because you never have to think about what happens first and what happens later – since no object is ever modified, the order in which things get executed does not matter.

True to this design principle, Wombat is a *functional* programming language, i.e. there are only expressions and never a sequence of statements that get executed one after the other. It is actually somewhat similar to XSLT. The only side effect necessary for model transformations is the creations of new model elements in the target models.

It turned out however that Wombat was useful for model *modifications* as well. Every model modification can be expressed as a transformation into a new model based on the same metamodel, but that can have a significantly bigger performance impact, and it may not be feasible if the modification is done e.g. on the current model inside a UML tool.

That required a couple of additional language elements for modifying and deleting model elements, but it left the key concepts intact. So Wombat still is a completely functional language, and it is helpful to have a functional programming mindset when using it (e.g. iterating over collections rather than using a procedural loop).

## Wombat scripts

### Script files

Wombat transformations are organized in script files. By convention, their file extension is \*.wbt.

Currently, Wombat requires an entire “program” to reside in a single file. Particularly, there are currently no mechanisms to call functions that are defined in a different file.

## Top-Level elements

Files can contain three kinds of top-level elements: function definitions, external extension definitions and internal extension definitions. These will be described in the sections “Functions” and “Extensions”, respectively.

Function and extension definitions can be arbitrarily mixed. The order in which the top-level elements in a Wombat script occur makes no semantic difference whatsoever.

## Comments

Wombat supports both line and block comments. The syntax is the same as in Java.

Two slashes (“//”) mark the rest of the line as a comment, and “/\*” makes Wombat ignore everything until the first occurrence of “\*/”.

## Whitespace

Whitespace does not have any semantic meaning. Whitespace is optional unless it is necessary to make clear how syntax elements are to be separated. SPACE, TAB, CR and NL are the whitespace characters.

A comment is semantically equivalent to whitespace.

## Identifiers

Identifiers in Wombat start with an uppercase or lowercase letter or an underscore (“\_”), followed by an arbitrary number of uppercase or lowercase letters, underscores or digits.

This differs from Java because in Wombat only 'a'-'z' are allowed as letters. However, it conforms to common practice for Java.

## Types

There are two kinds of types in Wombat, those that are built into the language and those that are defined by user-supplied metamodels.

### Built-in types

Wombat comes with support for a number of types readily built into the language.

First of all, there are the Java primitive types: *boolean*, *byte*, *short*, *char*, *int*, *long*, *float*, *double* and *string* (note the lowercase spelling! It helps avoid namespace conflicts with third-party

metamodels.) Although formally *string* is not a Java built-in type, it is a Wombat built-in type.

Then there are the collection types: *Set*, *List* and *Collection*, with *Set* and *List* as concrete types and *Collection* as an abstract super type of the two. There is special support for using collections (see section *Expressions*).

And lastly, there are the three special types *TYPE*, *ATTRIBUTE* and *OBJECT*. Wombat supports reflection on types and attributes, and *TYPE* and *ATTRIBUTE* are the corresponding types. They roughly correspond to `java.lang.Class` and `java.lang.reflect.Field`.

*OBJECT* is the abstract super type of all other types in Wombat.

All names of built-in types are reserved words in Wombat and may not be used as identifiers.

## User-supplied types

References from Wombat scripts to user-supplied types take the form

```
my_metamodel#my.package.MyType
```

where *my\_metamodel* is the name of the metamodel and *my.package.MyType* is the actual type name that the metamodel understands. It is permissible for a metamodel to map several names to the same type, and it is up to the metamodel whether it uses hierarchical dot-separated namespaces or not.

Wombat supports the notion of a *default metamodel*. This name of the default metamodel must however be configured from outside the script when Wombat is started. Wherever a type is referenced without giving a metamodel (like "#MyType"), the default metamodel is assumed.

## Functions

As explained in a previous section, Wombat is a functional programming language, so it is not a big surprise that the most important top-level language elements are *functions*.

## Syntax

A function is basically a name for an expression that can be called with parameters. The syntax for a function definition is as follows:

```
function_name (int param_1, int param_2):  
    "this is the result: " + (param_1 + param_2);
```

This piece of code defines a function named “function\_name” that takes two parameters of type int. After the colon, there is exactly one expression (see below) that defines the result of the function – in this case, a string that contains the arithmetic sum of the two parameters. The parameters are available by their names within the scope of this expression.

Since Wombat is a functional language, every function definition contains exactly one expression. Since there are no statements that could be executed sequentially, this is really the only way things could be.

The function definition ends with a semicolon.

### **Caching of results**

When transforming a graph, object identity in the target model is often important. It is for example a big difference if two cars have the same owner or two different owners who just happen to have the same name.

Wombat addresses this problem by ensuring that a function that is called with the same parameters always returns the *same* result. It does not just return two objects with the same values, but it really returns the same object!

So if two different objects need to reference the same object, this can be solved by creating a function that returns this object and then calling this function from both places.

This behavior is the primary difference between functions and extensions: functions have this behavior whereas extensions do not.

### **Calling a Wombat script**

A Wombat script is called by calling a function that is defined in this script. The cache used to handle object identity is maintained across several such function calls.

### **Lazy evaluation**

Object graphs are often cyclic. Creating such cyclic structures in a naïve manner leads to endless recursion. For example, if object a references object b and b has a reference to a, neither of the objects can be created completely without the other object being there.

Wombat takes care of such endless recursion through lazy evaluation. If a function creates and returns a new object, the object is created and cached as the result of the function, but the initializations of the fields are postponed.

## Expressions

Expressions are the core part of the Wombat language. A wide range of expressions is supported, some of them very similar to Java expressions and others very specific to Wombat.

This section presents them, ordering them by operator precedence and starting with the lowest.

### Initialization expression

An “initialization” is a special object. It groups an attribute and an expression and is needed as part of object creation.

```
a <- x.y + 25
```

On the left hand side of the arrow there is an expression that must evaluate to an object of type *ATTRIBUTE*. Within the scope of a *CREATE* expression – which is by far the most common place for an initialization expression – the names of the attributes are bound to the corresponding *ATTRIBUTE* objects, simplifying the syntax. It is however also possible to use an attribute literal instead.

On the right hand side of the arrow there is an arbitrary expression for which the only requirement is that its type must match that of the attribute.

### Conditional expression

A conditional expression is the functional equivalent of an if-else statement.

```
x < 5 ? 10 : 20
```

The syntax and semantic are identical to the corresponding Java expression.

Note that both the value for the “if” branch and the value for the “else” branch must be present. That is necessary because this is an expression and must therefore always evaluate to a value. But this is one of the places where functional programming differs from procedural programming.

### Chain expression

A chain expression is a list of expressions.

```
$print("Hello") -> $print(" ") -> $print("Arno")
```

They are all evaluated in the order they appear, but all results except for the last are discarded. A chain expression is useful to “execute” a sequence of expressions with side effects, opening the door to procedural programming within a functional language.

## Logical “or” expression

The logical “or” expression is identical to its Java equivalent.

```
i < 10 || y > 20
```

## Logical “and” expression

The logical “and” expression is identical to its Java equivalent.

```
i < 10 && y > 20
```

## Comparison expressions

The comparison expressions are syntactically identical to their Java equivalents.

```
x == 1  
x != 1  
x < 1  
x <= 1  
x > 1  
x >= 1
```

For primitive types, the semantic is identical to Java. For objects however, there are some subtle differences.

- *equals*. The “==” and “!=” operators are based on the equals method rather than sameness. They are however null-safe.
- *Comparable*. The other operators are applicable to all objects that implement the *Comparable* interface. Their implementation is null-safe.

## Arithmetic expressions

Wombat supports the arithmetic operators “+”, “-”, “\*”, “/” and “%” in a way that is identical to Java.

```
x + 2  
x - 2  
-x  
x * 2  
x / 2  
x % 2  
"The result is " + x
```

The “+” operator also serves the purpose of string concatenation.

## Identifier expressions

An identifier is interpreted by Wombat as a local variable and evaluates to the value bound to this variable.

```
x
```

If there are parentheses after the identifier, it stands for a function call.

```
x (15, "Hello")
```

It is possible to have several functions with the same name and different parameter types. In that case, a function call is matched to one of these function definitions according to the rules described in the section on *polymorphic invocation*.

## Property expressions

Wombat uses Java syntax for accessing a property or invoking a method of an object.

```
person.name  
car.owner.getAge()
```

There are however some subtle differences in the semantic.

- *Properties need not be Java fields.* It is up to the metamodel to know how to retrieve the value of a property. Oftentimes this will be done through a JavaBeans style get method rather than by directly accessing a field of the given name. The details of this are however specific to the metamodel implementation.
- *Polymorphic invocation of methods.* It is possible to have several methods with the same name but different parameter types. In such cases, the method to be called is selected based on the dynamic type of the parameter objects as described in the section on *polymorphic invocation*. This is different from the Java behavior where a method is picked at compile time based on the reference type of the parameter!

## Logical and bitwise “not” expression

The prefixed operators “!” and “~” denote the logical and bitwise negation of a value, just as they do in Java.

```
! (x + 7 < y)  
~1
```

## Extension call expression

An extension call expression calls an extension (see respective section) and evaluates to the result of this call (or *null* if the extension does not return a value).

```
$isPrimeNumber (x)  
$print ("Hello")
```

Extension calls are distinguished from function calls by a prefixed dollar sign.

It is possible to have several extension definitions with the same name and different parameter types. In this case, an extension call is matched to one of these function definitions according to the rules described in the section on *polymorphic invocation*.

## Type literal expression

Types are full-fledged objects in Wombat, so any type name is an expression that evaluates to the corresponding *TYPE* object.

```
a#Person
```

## Primitive literal expression

Wombat supports literals for the primitive types *boolean*, *int*, *double* and *string*.

```
true  
25  
-5.235  
"Hello \nArno"
```

These literals have the same syntax as their Java counterparts. Special characters in strings are escaped according to Java rules.

## Collection literals

A comma separated sequence of values denotes a *List* if it is enclosed in square brackets and a *Set* if it is enclosed in curly brackets.

```
[1, 2, "Hello again"]  
{1, 2, 1, 3, "..."}  

```

*List* and *Set* conform to the Java semantics: A list preserves the sequence of its values while a set removes duplicates without necessarily preserving sequence.

## Collection loops

Wombat has special syntax for applying an expression to all elements of a collection.

```
{2*x | x IN [1, 2, 3]}  
[2*x | x IN [1, 2, 3, 4, 5]]  

```

For every element of the collection on the right hand side, *x* is bound to this value and then the expression on the left is evaluated. All these values are collected into a new collection – a set if the surrounding brackets are curly and a list if they are square.

It is possible to add filtering expressions to the original expression, separated by commas. In that case, only those values go into the resulting collection that pass this filter.

```
[2*x | x IN [1, 2, 3, 4, 5], x > 1 && x y 4]
```

## Parentheses

Wombat supports parentheses to group subexpressions differently than their operator precedence would otherwise indicate, just as Java does.

```
5 * (x+1)
```

## Object creation

A create expression creates a new object in a given model, initializes it and evaluates to this value.

```
CREATE (a#Person IN targetModel :  
      {name <- x.name, firstName <- "Arno"})
```

The type of the object that is to be created is the value of an expression. In the simplest case, that is a type literal, but it can be any other expression as long as it evaluates to a *TYPE* object.

After the IN keyword there is an identifier that specifies into which model the new object should be created. It is possible to specify a “default creation model” when starting Wombat, in which case both the IN keyword and the identifier are optional.

After the colon, there is an expression that must evaluate to a collection of *initializations*. An initialization is a pair of an *ATTRIBUTE* object and an expression. Within the scope of a CREATE expression, the names of all attributes of a type are bound to the corresponding *ATTRIBUTE* values, making it simple to reference them. In the simplest and most common case, the collection of initializations is simply a collection literal, but any expression will do as long as it evaluates to a collection of initializations.

## Object modification

Wombat has an expression that just modifies the value of an attribute of an object.

```
SETATTRIB (car.owner, name, "Arno")
```

The first parameter specifies the object on which to change the attribute, the second is an identifier with the name of the attribute and the third is the new value the attribute should have.

This expression always evaluates to *null*. Note however that this is still an expression and not a statement!

## “Switch”

Wombat knows two flavors of *switch* statements, one based on Java syntax and the other as a replacement for cascaded if-else.

```
switch (age) {
  case 0 : "infant";
  case 6 : "school kid";
  case 18: "grown up";
  default: "whatever";
}

switch () {
  case age < 6: "pre school";
  case age < 18: "school";
  default:      "grown up";
}
```

There are two important differences to the way Java supports “switch”.

- The first variant supports switching over strings.
- Since Wombat “switches” are expressions rather than statements, they must always evaluate to a value. Therefore a “default” case must always be present.

## Collection arithmetic

Wombat supports the common operations to combine several collections.

```
UNION      ([1, 2, 3], {3, 4})
INTERSECT ([1, 2, 3], {3, 4})
WITHOUT    ([1, 2, 3], {3, 4})
```

Each of these operators takes an arbitrary number of collections as arguments. The type of the first collection determines whether the result is a set or a list.

The union of several collections is the collection containing all elements that any of the original collections contains.

The intersection of several collections contains those elements that all of these collections contain.

*WITHOUT* creates a collection with all elements that are in the first collection but not in any of the subsequent collections.

## Predicates on collections

Wombat supports predicates on collections.

```
FOREACH (x IN [1, 2, 3]: x < 3)
```

```
ANYOF (x IN [1, 2, 3]: x < 3)
```

*FOREACH* evaluates to *true* if and only if the conditional expression is *true* for every of the elements of the collection. *ANYOF* checks if it is *true* for at least one of the elements.

## Polymorphic invocation

There can be several functions, methods or extensions with the same name but different parameter types. In this case, Wombat picks the “best match” at execution time based on the type of the parameters that are actually passed to the call.

This behavior is different from Java where the decision is made at compile time based on the reference types of the parameters!

Often the way Wombat picks one of the functions is intuitive. But nonetheless it is good to understand how Wombat goes about this, so here is a description of the process.

In a first step, Wombat creates a shortlist of all functions for which the parameter types match the parameters that are actually passed.

In a second step, Wombat removes all those functions *f* from this shortlist for which there is a function *g* in the shortlist where the type of each formal parameter of *g* is a subtype of the corresponding formal parameter of *f*.

If this process results in a single resulting function, this function is called. If several functions remain, this is a runtime error because of the ambiguity.

An example will illustrate this. Let us assume that there are three functions *f* that take two parameters of different types.

```
f (int i, long j): ...;  
f (long i, int j): ...;  
f (long i, short j): ...;
```

If we call *f* with parameters of types *long* and *int*, only the second of the functions has a matching signature and is therefore called. This is the simplest case.

If we call *f* with parameters of types *long* and *short*, both the second and third function have formally matching signatures. The signature of the third function is however a complete specialization of the second function's signature, so the second function is removed from the shortlist and the third function is executed.

If we call *f* with parameters of types *int* and *short*, all three functions match. The second function is removed from the shortlist because the signature of the third is more specific in all parameters, so the first and third functions remain. Wombat can not decide which of them to call, triggering a runtime error.

## Extensions

Extensions are very similar to functions, the only two differences being that an extension call has a prepended dollar sign and that the results of extension calls are not cached.

There are two ways to define extensions in Wombat. One way is to use a Wombat expression similar to a function definition, the other is to import a Java class that contains the implementation of the extension.

### Internal definition

The syntax of an internal extension definition is identical to a function definition except for the keyword *EXTENSION* in front of it.

```
EXTENSION square (long l): l*l;
```

This extension can then be called as part of any expression:

```
$square (25) + 1
```

### External definition

The import of an externally defined extension also starts with the keyword *EXTENSION*, but after a colon there is the fully qualified name of the Java class containing the actual definition.

```
EXTENSION: my.package.MyExtensionClass;
```

All meta information about the extensions is contained in the Java class. For this reason, the Java class must implement the interface *ExtensionClassMetaDescription* in the package *org.openarchitectureware.wombat.impl.extension*. This interface has only one method, *getMetaDescriptions*, that returns a collection with meta data about all extensions defined in a class.

The abstract class *AbstractExtensionClass* provides a couple of useful convenience methods that simplify the most common cases of meta data for extensions.

It is possible to define an arbitrary number of extensions in a single Java class. In this way, semantically related extensions can be grouped together and can be imported together.