

The Flaka Manual

Wolfgang Häfelinger
häfelinger IT

Flaka, version **1.01**

February 23, 2010
document version 1.0

Introduction

In the world of [Java](#), build scripts are traditionally written in [Ant](#), recently also using [Maven](#).

Writing a project's build script is serious business. And so it is when using Ant. Ant does not provide you with any abstraction how the project needs to be build. There is no underlying logic. In fact you, the author, need to know what exactly needs to be done. Step by step. What's more, you have to use a rather unfriendly, sometimes even hostile, [XML](#) syntax. By default Ant provides you a rather large list of *tasks*¹ and *types*². Types are used to create data objects which can be used to feed tasks. Tasks are sequentially arranged to larger work units, called *targets*. Finally there are properties to establish a line of communication between work units and tasks.

Flaka 1.01
häfelinger IT

2/99

So writing a build script in Ant is like writing a Shell script where you have all those small masterpieces like `mkdir`, `cp`, `tar` available while decent control structures, like `if` and `for`, have been stripped off. This forces you, the author of a build script, to think in new categories and in rather creative ways. Ant scripts are therefore usually large, often work on the author's machine only and each author uses a different set of target and property names making it difficult to understand someone else's script.

Maven on the other side provides a high abstraction of building a project. Instead of describing how a project needs to be build, just describe the project detail and let Maven figure out what needs to be done. This is probably the reason why Maven got so much attention recently. Despite better knowledge I wrote that Maven figures out what needs to be done. That's actually not quite true. In fact, Maven works generally fine when following conventions imposed by Maven. When not en route, Maven gets difficult as well. But even when following conventions, the number of options in Maven are, with the event of Maven's second incarnation, endless and question the idea of a declarative approach³.

At the end, I found myself using Ant again.

However, what I am missing in Ant is the full power of a programming language.

¹ <http://ant.apache.org/manual/coretasklist.html>

² <http://ant.apache.org/manual/conceptstypeslist.html>

³ Have a look at Maven's [POM](#) and checkout the never ending series of XML tags possible

Yes, I want to have conditionals, looping constructs and a decent exception handling. I want to have variables which I can set or remove for pleasure. I don't want to be restricted that such variables may carry strings only. Any data object must be allowed. Eventually I need a nice expression language to retrieve and calculate data in a simple yet elegant way. There is no need to have each and everything expressed in XML. And then I want to have some kind of higher abstraction which does the right thing most of the time.

Flaka 1.01
häfelinger IT

In summary, this is what Flaka is all about:

3/99

- Control Structures
- Expressions
- Framework to do the *right* thing

These pillars are Flaka's approach to simplify the process of writing a build script with Ant. You are by no means forced to use all or any of those pillars. You can for example just use the programming tasks with or without EL while you don't need to get in touch with Flaka's dependency handling instruments and neither with the framework.

Where to go from here?

- [Download](#) Flaka and read the [installation page](#).
- Jump right into chapter [Overview](#) to get some ideas about elements provided by Flaka.
- Make sure to consult chapter [EL](#). It contains a lot of information on this enormous useful extension.
- Have a closer look in the reference part of this manual for all the gory details.
- Start writing build scripts using Flaka and give [feedback](#).

Flaka 1.01
häfelinger IT

4/99

About This Manual

Which Flaka version?

This manual corresponds with **Flaka 1.01**.

Flaka 1.01
häfelinger IT

Overall Structure

There are four parts:

5/99

1. An overview over Flaka's concepts are presented in [Part I](#).
2. [Part II](#), is the reference part for all elementary control structures and for all of EL's gory details.
3. In [Part III](#), special purpose tasks are presented. Most of this tasks are implemented with the help of tasks and concept shown in Part I.
4. The last part, [Part IV](#), is about the installation of Flaka.

Conventions

Ant build file examples show a mix of tasks provided by Flaka and by Ant. Ant task do not require a namespace while those provided by Flaka do. Flaka's namespace is

```
| antlib:it.haefelinger.flaka
```

and within this manual, the abbreviation `c` will be used for this namespace. Therefore it becomes easy to see who is the provider of a task:

```
| <echo> This is Ant </echo>  
| <c:echo> and this is Flaka's echo</c:echo>
```

Thus all build file snippets shown assume that the build file contains the following XML namespace declaration:

```
| <project xmlns:c="antlib:it.haefelinger.flaka" ..>  
|   <!-- build script example -->  
| </project>
```

Part I, Overview

Flaka 1.01
häfelinger IT

6/99

An Expression Language

The [Java Unified Expression Language](#) is a special purpose programming language offering a simple way of accessing data objects. The language has its roots in Java web applications for embedding expressions into web pages. While the expression language is part of the JSP specification, it does in no way depend on the JSP itself. To the contrary, the language can be made available in a variety of contexts.

Flaka 1.01
häfelinger IT

7/99

One such context is Ant scripting. Ant makes it difficult to access data objects. For example, there is no way of querying the underlying data object for the base folder (the folder containing the build script). All that Ant offers is the path name of this folder as *string* object. This makes it for example rather cumbersome to report the last modification time of this folder. With the help of EL (short for *Expression Language*) this becomes an easy task:

```
<c:echo>
  ;; basedir is a standard Ant property
  basedir is ${basedir}

  ;; report last modification time (as Date object)
  was last modified at #{ '${basedir}'.tofile.mtime }

  ;; dump the full name of this build file
  ;; where 'ant.file' is a standard property
  this is #{property['ant.file']} } reporting!
</c:echo>
```

Being executed, this snippet produces something like

```
[c:echo] basedir is /projects/flaka/test
[c:echo]
[c:echo] was last modified at Mon Mar 09 13:52:29 CET
      2009
[c:echo]
[c:echo] this is /projects/flaka/test/tryme.xml
      reporting!
```

Notice that Flaka's [echo](#) task has been used for this illustration because [EL](#) is by default only available on Flaka tasks. If [Ant's standard echo](#) task is used, all

`#{. .}` constructs are left as they are. It is however possible and recommended to turn [EL](#) on for *all* tasks ⁴].

The next code example shows another *EL in action* sample. The programming problem is to list all unreadable (sub)folders in a certain folder - here being the *root* folder:

Flaka 1.01
häfelinger IT

8/99

```
<c:let>
  ; The root folder as file object. Notice that the
    right side of each
  ; assignment is an EL expression.
  root = './'.tofile
  ; This creates an empty list
  list = list()
</c:let>

<c:for var="file" in=" root.list ">
  <c:when test=" file.isdir and not file.isread ">
    <c:let>
      ;; The condition above is a EL expression and so
        is this
      ;; 'append' function.
      list = append(file,list)
    </c:let>
  </c:when>
</c:for>

<c:echo>
  ;; how many unreadable directories ??
  There are #{size(list)} unreadable directories in #{
    root}.
  And here they are #{list}.
</c:echo>
```

Executed on MacOS 10.5.6 (aka "Leopard") gives:

```
[c:echo] There are 2 unreadable directories in /.
[c:echo] And here they are [/.Trashes, /.Spotlight-
  V100].
```

⁴ [See [how to enable EL](#) for details

Have a look at [EL](#) for further EL examples and details.

Flaka 1.01
häfelingen IT

9/99

Conditionals

With standard Ant, task `condition` is used to set a property if a condition is given. Then a macro, task or target can be conditionally executed by checking the existence or absence of that property (using standard attributes `if` or `unless`). Flaka defines a couple of control structures to handle conditionals in a simpler way.

Flaka 1.01
häfelinger IT

10/99

when and unless

Task `when` evaluates an `EL` expression. If the evaluation gives `true`, the sequence of tasks are executed. Nothing else happens in case of `false`.

```
<c:when test=" expr ">
  -- executed if expr evaluates to true
</c:when>
```

The logical negation of `when` is task `unless` which executes the sequence of tasks only in case the evaluation of `expr` returns `false`.

```
<c:unless test=" expr ">
  -- executed if expr evaluates to false
</c:unless>
```

The body of `when` and `unless` may contain any sequence of tasks or macros (or a combination of both).

choose

Task `choose` tests each `when` condition in turn until an `expr` evaluates to `true`. It executes then the body of that `when` condition. Subsequent `whens` are then not further tested (nor executed). If all expressions evaluate to `false`, an optional `catch-all` clause gets executed.

```
<c:choose>
  <when test="expr_1">
    -- body_1
  </when>
  ..
  <otherwise> -- optional_
```

```
    -- catch all body
  </otherwise>
<c:/choose>
```

switch

A programming task often seen is to check whether a (string) value matches a given (string) value. If so, a particular action shall be carried out. This can be done via a series of *when* statements. The nasty thing is to keep track of whether a value matched already. Flaka provides a handy task for this common scenario, the `switch` task:

```
<c:switch value=" 'some string' ">
  <matches re="regular expression or pattern" >
    -- body_1
  </case>
  ..
  <otherwise> -- optional
    -- catch all body
  </otherwise>
</c:switch>
```

Each case is tried in turn *to match* the string value (given as [EL](#) expression). If a case matches, the appropriate case body is executed. If it happens that no case matches, then the optional default body is executed. To be of greater value, a regular expression or pattern expression can be used in a case condition.

Flaka 1.01
häfelinger IT

11/99

Repetition

Flake has a looping statement. Use task `for` to iterate over a *list* of items. Use `break` and `continue` to terminate the loop or to continue the loop with the next item.

```
<c:for var=" name " in=" '' .tofile.list ">
  -- sequence of task or macros
  -- used <c:continue /> to continue ; and
  -- <c:break /> to stop looping
  -- use #{name} to refer to current item (as shown
     below)
  <c:echo>#{name}</c:echo>
</c:for>
```

Flake 1.01
häfelinger IT

12/99

Attribute `in` will be evaluated as [EL](#) expression. In the example above, that [EL](#) expression is `'' .tofile.list` which, when evaluated, creates a list of all files in the folder containing the current build script. To understand the expression, have a look at [properties](#) of a string] and [properties](#) of a file.

Exception Handling

Flaka has been charged with exception handling tasks.

trycatch

Flaka contains a task to handle exceptions thrown by tasks, `trycatch`. This task implements the usual *try/catch/finally* trinity found in various programming languages (like in Java for example):

Flaka 1.01
häfelinger IT

13/99

```
<c:trycatch>
  <try>
    -- sequence of task or macros
  </try>
  <catch>
    -- sequence of task or macros
  </catch>
  <finally>
    -- sequence of task or macros
  </finally>
</c:trycatch>
```

Element *try*, *catch* and *finally* are all optional or can appear multiple times. If *catch* is used without any argument, then that catch clause will match any **build exception**. To differentiate between different exception types, *catch* can additionally be used with a *type* and *match* argument. The former can be used to select a particular exception type (like a `java.lang.NullPointerException`), the latter can be used to select an exception based on the message carried. Both arguments are interpreted as pattern expression. For example:

```
<c:trycatch>
  <try>
    ..
    <fail message="#PANIC!" unless="ant.file"/>
    ..
  </try>
  <catch match="##PANIC!*">
    <echo>Ant initialization problem!!</echo>
    <fail/>
  <catch type="java.lang.*">
    -- handle Java runtime problems
```

```
</catch>
<catch>
  -- handle all other build exceptions
</catch>
</c:trycatch>
```

Property *ant.file* is a standard Ant property that should always be set. If not, there's something seriously wrong and it does not make much sense to continue. Use attribute *type* to catch (runtime) exceptions thrown by the underlying implementation.

Flaka 1.01
häfelinger IT

14/99

throw

Task [throw](#) throws a (build) exception.

```
<c:throw [var="sym"] />
```

This task can also be used to rethrow an existing exception.

Procedures

Ant Macros

The (almost) equivalent of a procedure is a macro in Ant and Flaka. For example:

```
<macrodef name="hello">
  <attribute name="msg" />
  <element name="body" implicit="true" />
  <sequential>
    <body />
  </sequential>
</macrodef>
```

Flaka 1.01
häfelinger IT

15/99

Once defined, simply use it:

```
<hello msg="Hello, world!">
  <echo>@{msg}</echo>
</hello>
```

This macro evaluates into

```
<echo>Hello, world!</echo>
```

which eventually prints the desired greeting.

Macros are a standard feature of Ant.

Sequencing

To evaluate a sequence of expressions (tasks or macros) where only one expression is allowed, use [Ants sequential task](#):

```
<sequential>
  -- any sequence of tasks or macros
</sequential>
```

Note that *sequential* returns nothing. Use properties to communicate with the caller if necessary.

Flaka Glossary

A compilation of words and their meaning in Flaka.

Continuation Lines

A continuation line is a sequence of characters ending in `\NL` and not in `\\NL` (where `NL` is the line ending characters `CR`, `LF` or `LF`). Tasks supporting continuation lines will accumulate the content of such a line with the (accumulated) content of the following line. The continuation character and the line ending characters are not accumulated.

```
| a \  
| b\  
| c\  
|
```

Defines two accumulated lines: (1) `a b\
c` and (2) `c`.

Property Reference

A reference to a [property](#) is written as `${. .}`. Property references are handled by the Ant property handler. If not changed, then `${x}` will be replaced by the value of property `x` if such a property exists. Otherwise, the reference will be left as is.

Expression Reference

A reference to an [EL](#) expression is written as `#{. .}`. [EL](#) is not part of Ant and can thus only be handled by certain tasks. References may appear in attribute values or in text elements. Not all attributes can handle EL references and neither all text elements. If an attribute or text element can handle EL references, it is specifically mentioned.

Base Folder

Relative files are usually meant to be relative to the current working directory. Not so in Ant, where a file is relative to the folder containing the build script of the current project. This folder is called the base directory or base folder. Ant defines property `basedir` to contain the (absolute) path name of this folder.

Flaka 1.01
häfelinger IT

16/99

When using [EL](#) expressions you can use the empty string to create the base folder as file object, like in ``'.tofile``.

See also [built-in-props](#) for a list of standard Ant properties.

Flaka 1.01
häfelinger IT

17/99

Part II, Elementary

Flaka 1.01
häfelingen IT

18/99

EL

The gory details of *EL* are laid out in the [the official JSR 245 specification](#) and are not repeated here. In short however, *EL* lets you formulate [programming expressions](#) like

```
7 * (5.0+x) >= 0      ;; 1
a and not (b || false) ;; 2
empty x ? 'foo' : x[0] ; 3
```

Flaka 1.01
häfelinger IT

19/99

The expression in line (1) is a algebraic while (2) contains a boolean expression. The result of (1) depends on the resolution of variable *x* and similar does (2) on *a* and *b*. Line (3) shows the usage of two built-in [operators](#) (see below for details).

Such expressions can hardly expressed in pure Ant. However, a even more useful concept of EL is

[.] the evaluation of a model object name into an object, and the resolution of properties applied too objects in an expression (operators `.` and `[]`).

— JSR 245 specification *Page 22*

EL provides a nice way of accessing underlying data objects. Assume that we have a EL variable named *d* and it is (somehow) associated with a Java object. Then we can query object properties in an expression like `d._name_` or `d[name]`. Two examples:

- Assume that object *d* is a Java File object. Then `d.mtime` would return the last modification time of that file as object of type Date.
- Assume that object *d* is a Java Map object. Then `d.foobar` would query a value in that map object using `foobar` as key.

The rest of this chapter introduces relevant details of EL in order to use it within Flaka.

Globally Enabling EL

By default can EL expressions can only be used in tasks which are EL aware. This are all tasks provided by Flaka but not those provided by Ant or any contribution to Ant. It is possible though to **enable EL** on a global level - i.e. for all tasks. To enable handling of EL references on a global level on all tasks, types or macros and vector independent, use task [install-property-handler](#):

```
| <c:install-property-handler />
```

Flaka 1.01
häfelinger IT

20/99

EL References

Those *not* familiar with the specification of [EL](#), [JSP](#) or [JSF](#) may safely skip this section. All other please read on, cause the implementation of EL has slightly be changed ⁵.

For those familiar, the *term EL expression* is used in a slightly different way in this documentation than in the specification. According to the specification, `{ expr }` is an EL expression.

Not so in this documentation. Here only *expr* - the inner part - is considered a *EL expression* while `{ expr }` is considered a EL expression *reference*.

A reference is used in contexts which should be partially evaluated. Take task [echo](#) as example. Clearly, when writing

```
| <c:echo>  
| I said 'Hello world'!  
| </c:echo>
```

we expect an output exactly as written. It would be nice to indicate however, that we want to have a part of the input evaluated as EL expression. This and only this is what `{ expr }` is good for:

```
| <c:let>  
| hello = 'Hello world'  
| </c:let>
```

⁵ EL has its roots in the context of Java Web Development and some specification details do not make sense when EL is used in a different domain content

```
<c:echo>
  I said '#{ hello }'!
</c:echo>
```

In other contexts, like in `<c:when test=" condition " />`, partial evaluation of *condition* does not make sense. Instead the whole *condition* is expected to be a EL expression. Wrapping *condition* with `#{` and `'}`` is just an unnecessary overhead which makes reading difficult.

Flaka 1.01
häfelinger IT

21/99

As an example, assume that we want to check whether property `foobar` exists. This can be expressed using `has.property.foobar`. Nevertheless, in JSP you would have to write

```
<c:when test=" #{ has.property.foobar } " />
```

In Flaka you can omit `#{` and `'}`` and go ahead with

```
<c:when test=" has.property.foobar " />
```

However, the EL expression itself is subject to partial evaluation. So you may want to write

```
<c:when test=" has.property.#{ name } " />
```

to dynamically check for the existence of property *name*. And of course you may also use a combination of Ant property and EL references like in

```
<c:when test=" has.#{category}.#{ name } " />
```

Nested References

Nested references are not supported. The following reference is therefore illegal

```
#{ item[ #{ index } ] }
```

Handling of `#{..}` and `${..}`

In general, Ant property references `${..}` are resolved before EL references. Thus the following works fine:

```
| has.property [ '${somename}' ]
```

The Great Escape

This section is about how to stop a EL reference from being evaluated and treated as text instead:

- Use character backslash like in `\#{abc}` ; or use this rather awkward
- `#{'#{'}abc}` construct.

Both variants have the same result, the string `#{abc}`.

Data Types

EL's data types are integral and floating point numbers, strings, boolean and type `null`. Example data values of each type, except type `null`, are given above (1-3). Type `null` has once instance value also named `null`. While `null` can't be used to formulate an expression, it is important to understand that the result of evaluating an expression can be `null`. For example, the evaluation of a variable named `x` is the data object associated with that name. If no data is associated however (i.e. if `x` is undefined), then `x` evaluates to `null`.

Strings

A EL string starts and ends with the same quotation character. Possible quotation characters are single the quote `'` and double quote `"` character. If string uses `'` as quotation character, then there is no need to *escape* quotation character `"` within that string. Thus the following strings are valid:

```
| "a'b"    --> a'b  
| 'a"b'    --> a"b
```

If however the strings quotation character is to be used within the string, then the quotation character needs to be escaped from its usual meaning. This is done by prepending character backslash:

```
| "a\"b"    --> a"b  
| 'a\'b'    --> a'b
```

To escape the backslash character from its usual meaning (escaping that is), escape the backslash character with a backslash:

```
"a\\" --> a\  
'a\'' --> a\  
|
```

Other characters than the quotation and backslash character can't be escaped. Thus

```
"a\bc" --> a\bc, NOT abc  
|
```

However, a escaped backslash evaluates always into a single backslash character:

```
"a\\b" --> a\b, NOT a\\b  
|
```

This rules allow for an easy handling of strings. Just take an quotation character. Then, escape any occurrences of the quotation and escape character within the string to preserve the original input string.

Here are some further examples strings:

```
"abc" -- abc  
'abc' -- abc  
"a'c'" -- illegal  
"a'c" -- a'c  
'a\'c' -- a'c  
'a\bc' -- a\bc  
'a\\bc' -- a\\bc  
'a\"bc' -- a\"bc  
'a\\"bc' -- a\\"bc  
'ab\' -- illegal  
'ab\\' -- ab\  
|
```

Operators

This are the most important operators defined in EL:

- `empty` checks whether a variable is empty or not and returns either `true` or `false`. It is important to understand that `null` is considered empty.

- condition operator `c ? a : b` evaluates `c` in a boolean context and returns the evaluation of expression `a` if `c` evaluates to `true`; otherwise `eval(b)` will be the result of this operator.
- `.` and `[]` are property operators described in [section Properties](#) below.
- logical operators `not`, `and` and `or`
- relational operators `==`, `!=`, `<`, `>`, `<=` and `>=` (resp. `eq`, `ne`, `lt`, `gt`, `le` and `ge`).
- usual arithmetic operators like `+`, `-`, `*`, `/`, `mod` and `div` etc.

Flaka 1.01
häfelinger IT

24/99

Properties

Every data object in *EL* may have properties associated. Which properties are available has not been standardized in the [specification](#). In fact, this depends heavily on the underlying implementation and usage domain. What *EL* specifies however, is how to query a property:

```
| a.b.c
```

This expression can be translated into pseudo code as

```
| (property 'c' (property 'b' (eval a)))
```

which means that first variable `a` is evaluated, then property `b` is looked up on the evaluation result (giving a new evaluation result) and finally `c` is looked up giving the final result.

Perhaps the most important point to notice is looking up a property on `null` is not an error but perfectly legal. No exception gets raised and no warning message generated. In fact, the result of such a operation is just `null` again.

From a practical point a question might be asked how to query a property which happens to contain the dot (`.`) character. In `a.b.c` example shown above, how would we lookup property `b.c` on `a`? Operator `[]` comes to rescue:

```
| a['b']           => a.b
| (a['b'])['c']   => a.b.c
| a['b']['c']     => a.b.c
```

```

a[b]           => can't be expressed using '.'
a[b.c]        => neither this ..
a['b.c']      => query property 'b.c' on a

```

So far, properties don't seem of any good use. The picture changes perhaps with this example:

```

'abc'.toupper      => 'ABC'
'abc'.length*4    => 12
'abc')['tofile'].mkdir => true/false

```

Flaka 1.01
häfelinger IT

25/99

The last example demonstrates that there might also be [side effects](#) querying a property. In the example above, which is specific for Flaka, a directory `abc` gets created and the whole expression evaluates to `true` if the directory could get created and `false` otherwise.

See further down which properties are available on various data types.

Implicit Objects

Properties are good to query the state of data objects. The question is however, how do we get a data object to query in the first place? To start with *something*, EL allows the implementation to provide *implicit* objects and [top level functions](#) (see below).

The following implicit objects are defined by Flaka:

Implicit Object	Type	Description
<i>name</i>		If <i>name</i> is not a predefined name as listed in the rest of this table, then <i>name</i> will be the same as <code>var [name]</code> , i.e. <i>name</i> will resolve to the object associated with variable <i>name</i> .
<code>project</code>		Ants underlying project object. It can be used to query the default target, base folder and other things. If you want to query properties, references, targets, tasks, taskdefs, macrodefs, filters etc., use appropriate implicit object instead.
<code>property</code>		Use this object to query project properties.

var		A object containing all project references.
reference		Same as var
target		Use this object to query a target
taskdef		Query taskdefs
macrodefs		Macros
tasks		Either taskdef or macrodef. Macros are specialized task and thus same the same namespace.
filter		A object containing all filters defined in this project.
e	double	The mathematical number <i>e</i> , also known as Euler's number .
pi	double	The number <i>PI</i>

Flaka 1.01
häfelinger IT

26/99

An example for an EL expression fetching property foo is:

```
property.foo
project.properties.foo
```

Similar, a variable named foo is fetched like

```
foo -- (1)
var.foo -- (2)
reference.foo -- (3)
project.references.foo -- (4)
```

Functions

EL also allows the implementation to provide top level functions. The following sections describe functions provided by Flaka. Some functions take an arbitrary number of arguments (inclusive no argument at all). This is denoted by two dots (. .). An example of such a function is `list(object..)` which takes an arbitrary number of object to create a list.

Function	Type	Meaning
<code>typeof(object)</code>	string	The type of object, int, string, file etc

<code>size(object)</code>	int	Returns the object's size. The size of the object is given by the number of entities it contains. This is 0 (zero) for all primitive types. Otherwise the size is determined by an underlying <code>size()</code> method or <code>size</code> or <code>length</code> attribute of the object in question.
<code>sizeof(object)</code>	int	same as <code>size(object)</code> , see above
<code>null(object)</code>	bool	Evaluates to <code>true</code> if object is the <code>nil</code> entity; otherwise <code>false</code> . This function can be used to check whether a reference (<code>var</code>) or property exists. Operator <code>empty</code> can't be used for this task, cause <code>empty</code> returns <code>true</code> if either not existing or if literally <code>empty</code> (for example the empty string).
<code>file(object)</code>	File	Creates and returns a file object out of object. If object is already a file, the object is simply returned. Otherwise, the object is streamed into a string and that string is taken as the files path name.
<code>concat(object..)</code>	string	Creates a string by concatenating all stringized objects. If no object is provided, the empty string is returned.
<code>list(object..)</code>	list	Returns a list where the lists elements consists of the objects provided. If no objects are provided, the empty list is returned.
<code>append(object..)</code>	list	This function is similar to <code>list</code> . It takes the objects in order and creates a list elements out of them. If a object is a list, then elements of that list are inserted instead of the list object itself. For example <code>append('a',list('b'),'c')</code> evaluates to <code>list ('a', 'b', 'c')</code>

Flaka 1.01
häfelinger IT

27/99

Some mathematical functions are defined as well:

<code>sin(double)</code>	double	The mathematical sine function
<code>cos(double)</code>	double	The mathematical cosine function

tan(double)	double	The mathematical tangent function
exp(double)	double	The mathematical exponential function, e raised to the power of the given argument
log(double)	double	The mathematical logarithm function of base e
pow(double)	double	Returns the value of the first argument raised to the power of the second argument.
sqrt(double)	double	Returns the correctly rounded positive square root of a double value.
abs(double)	double	Returns the absolute value of a double value.
min(double, double)	double	Returns the smaller of two double values.
max(double, double)	double	Returns the larger of two double values.
rand()	double	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

Flaka 1.01
häfeling IT

28/99

Available Properties

In general properties are mapped as *attribute* on the underlying data object. In Java, every *getX* method taking no arguments identifies property *x*. As an example, assume that we have

```
public class Foo {
    public .. getBar() { .. }
}
```

then an data object of type *Foo* will have property *bar* and thus the following expression *x.bar* would eventually call *Foo.getBar()* assuming that *x* evaluates to an object of type *Foo*. Such properties are the **natural** properties of a type.

Primitive Types

Primitive data types (int, double, bool, null) have no properties.

List and Arrays

Besides their *natural* properties (see discussion above) are *index* properties available:

```
| list('a','b')[1] => 'b'
```

Negative indexes are currently not supported. If an index is specified and not existing element, `null` is returned.

Flaka 1.01
häfelinger IT

29/99

String Properties

Besides *natural* properties (see discussion above) are the following properties supported:

Property	Type	Description
length	int	number of characters in this string
size	int	same as property length
tolower	string	return this string in lowercase characters only
toupper	string	return this string in uppercase characters only
trim	string	remove leading and trailing whitespace characters
tofile	file	create a file based on this string; the so created will be relative to the current build files base folder if the strings value does not denote a absolute path. Furthermore, the empty string will create a file object denoting the projects base folder (i.e. the folder containing the build script currently executed). Notice that <code>.</code> and <code>..</code> denote absolute paths, not relative ones.

File Properties

Files and folders is Ants bread and butter. A couple of properties are defined on file objects to simplify scripting (see below). Most important is however how to *get* a file object in the first place. This is most easily done by using string property `tofile`:

```
| 'myfolder'.tofile
```

In this example of an EL expression, string `myfolder` is converted in a `File` object using property `tofile`. Then, having a file object at your finger tips, the following properties are available:

Property	Type	Description
<code>absoluteFile</code>	File	The absolute form of this abstract pathname
<code>absolutePath</code>	String	The absolute form of this abstract pathname
<code>canonicalFile</code>	File	The canonical form of this abstract pathname
<code>canonicalPath</code>	String	The canonical form of this abstract pathname
<code>delete</code>	bool	deletes the file or folder (true); false otherwise
<code>exists</code>	bool	check whether file or folder exists
<code>isdir</code>	bool	check whether a folder (directory)
<code>isfile</code>	bool	check whether a file
<code>ishidden</code>	bool	check whether a hidden file or folder
<code>isread</code>	bool	check whether a file or folder is readable
<code>iswrite</code>	bool	check whether a file or folder is writable
<code>length</code>	int	same as <code>size</code>
<code>list</code>	File	array of files in folder
<code>mkdir</code>	bool	creates the folder (and intermediate) folders (true); false otherwise
<code>mtime</code>	Date	last modification date
<code>name</code>	String	The basename
<code>parent</code>	File	parent of file or folder as file object
<code>path</code>	String	abstract pathname into a pathname string.
<code>size</code>	int	number of bytes in a (existing) file; 0 otherwise
<code>toabs</code>	File	file or folder as absolute file object
<code>tostr</code>	String	file name as string object
<code>touri</code>	URI	file as URI object
<code>tourl</code>	URL	file as URL object

Flaka 1.01
häfelinger IT

30/99

Matcher Properties

A *matcher object* is created by task `switch` if a regular expression matches a input value. Such a matcher object contains details of the match like the start and end position, the pattern used to match and it allows to explore details of

capturing groups (also known as `_` marked subexpression).

Property	Type	Description
<code>start</code>	int	The position within the input where the match starts.
<code>s</code>	int	Same as <code>start</code>
<code>end</code>	int	The position within the input where the match ends (the character at <code>end</code> is the last matching character)
<code>e</code>	int	Same as <code>end</code>
<code>groups</code>	int	The number of capturing groups in the (regular) expression.
<code>size</code>	int	Same as <code>groups</code>
<code>length</code>	int	Same as <code>groups</code>
<code>n</code>	int	Same as <code>groups</code>
<code>pattern</code>	string	The regular expression that was used for this match. Notice that glob expressions are translated into regular expressions.
<code>p</code>	string	Same as <code>pattern</code>
<code>i</code>	matcher	The matcher object for <code>i</code> 'th capturing group. See task switch for examples.

Flaka 1.01
häfelinger IT

31/99

Evaluating in a boolean context

When evaluation a `expr` in a string context, a string representation of the final object is created. Similar, when a evaluation in a boolean context takes place, a conversion into a boolean value of the evaluated object takes place. The following table describes this boolean conversion:

evaluated object type	true	false
<code>file</code>	if the file exists	false otherwise
<code>string</code>	if string is empty	false otherwise
<code>null</code>	never	always
<code>boolean</code>	if true	otherwise
<code>other</code>	always	never

break

A task mirroring a break statement. When used within a [for](#)-loop, the loop will be terminated. If this task is used outside of a [for](#)-loop, a build exception will be thrown.

Flaka 1.01
häfelinger IT

32/99

```
<c:for var="i" in=" list(1,2,3,4,5,6) ">
  <c:echo>i = #{i}</echo>
  <c:when test=" i == 3 ">
    <c:break />
  </c:when>
</c:for>
```

Being executed, the following will be dumped on stdout:

```
[c:echo] i = 1
[c:echo] i = 2
[c:echo] i = 3
```

Attributes

Attribute	Type	Default	EL	Description
test	string	-	expr	Terminate loop when EL expression evaluates to true
if	string	-	#{}	Terminate if property exists
unless	string	-	#{}	Terminate if property does not exist

Behaviour

When used without any attributes, the surrounding [for](#) or [while](#) loop will terminate at once. When used with attributes, then the loop will terminate if at least one attribute evaluates to true. Otherwise, the loop will not be terminated and continues as usual.

The example given above can thus be shortened to

```
<c:for var="i" in=" list(1,2,3,4,5,6) ">
  <c:echo>i = #{i}</echo>
```

```
<c:break test=" i == 3 " />  
</c:for>
```

Further Links

- [Javadoc](#)
- [Source](#)

Flaka 1.01
häfelinger IT

33/99

choose

A task implementing a series of *ifelse* statements, i.e. a generalized *if-then-else* statement.

Attributes

Attribute	Type	Default	EL	Description
<i>when.test</i>	string	false	=	A EL condition. When <code>true</code> corresponding clause will be executed.
<i>unless.test</i>	string	true	=	A EL condition. When <code>false</code> corresponding clause will be executed.
debug	boolean	false	=	Turn on extra debug information.

Flaka 1.01
häfelinger IT

34/99

Elements

Element	Cardinality	Description
when	infinite	To be executed if condition evaluates to <code>true</code>
unless	infinite	To be executed if condition evaluates to <code>false</code>
otherwise	[0,1]	To be executed if no <code>when</code> or <code>unless</code> clause got executed
default	[0,1]	Synonym for <code>otherwise</code>

Behaviour

Each `when` and `unless` clauses conditions are evaluated in order given until a clause gets executed. Then, further processing stops ignoring all further elements not taken into account so far. If no `when` or `unless` clause got executed, then a present `otherwise` or `default` clause gets executed.

The shortest possible `choose` statement is

```
| <c:choose />
```

Its useless and does nothing, its completely harmless.

The following example would execute all macros or tasks listed in the otherwise clause cause no when or unless clause got executed.

```
<c:choose>
  <otherwise>
    <!-- macros/tasks -->
  </otherwise>
</c:choose>
```

Flaka 1.01
häfelinger IT

35/99

This would execute all macros and tasks listed in the otherwise clause since no when clause got executed.

```
<c:choose>
  <when test=" true == false" >
    <echo>new boolean logic detected ..</echo>
  </when>
  <unless test=" 'mydir'.tofile.isdir ">
    <echo> directory mydir exists already </echo>
  </when>
  <otherwise>
    <echo> Hello,</echo>
    <echo>World</echo>
  </otherwise>
</c:choose>
```

Further Links

- [Javadoc](#)
- [Source](#)

continue

A task mirroring a continue statement. When used within a [for](#)-loop, the loop will be continued with the next loop item (i.e. any statements after task continue are ignored). If this task is used outside of a for-loop, a build exception will be thrown.

Flaka 1.01
häfelinger IT

36/99

```
<c:for var="i" in=" list(1,2,3,4,5,6) ">
  <c:when test=" i > 3 ">
    <c:continue />
  </c:when>
  <c:echo>i = #{i}</echo>
</c:for>
```

This would print:

```
[c:echo] i = 1
[c:echo] i = 2
[c:echo] i = 3
```

Attributes

Attribute	Type	Default	EL	Description
test	string	-	expr	Continue loop when EL expression evaluates to true
if	string	-	#{}	Continue if property exists
unless	string	-	#{}	Continue if property does not exist

Behaviour

When used without any attributes, the surrounding [for](#) or [while](#) be continued while following tasks or macros are ignored in the current iteration step. When used with attributes, then the loop will be continued if at least one attribute evaluates to true. Otherwise, the subsequent tasks or macros are executed.

The example given above can thus be shortened to

```
<c:for var="i" in=" list(1,2,3,4,5,6) ">
```

```
<c:continue test=" i > 3 " />  
<c:echo>i = #{i}</echo>  
</c:for>
```

Further Links

- [Javadoc](#)
- [Source](#)
- Task [for](#)
- Task [break](#)

Flaka 1.01
häfelingen IT

37/99

echo

Ant has an echo task to dump some text on a screen or into a file. A problem with this task is, that the output produced is rather fragile when it comes to reformatting your XML source. Here is a simple example.

```
| <echo>foobar</echo>
```

When executed by Ant, this dumps

```
| [echo] foobar
```

However, one day you reformat your XML build file ⁶ and you end up in

```
| <echo>  
|   ...foobar  
| </echo>
```

Notice the usage of character . (dot) in this example and the rest of this (and only this) chapter to visualize a *space* ⁷ character. If you execute this, you will get

```
| [echo]  
| [echo] ...foobar  
| [echo]
```

This is definitely not what you had in mind.

Task `<c:echo/>` is an extension of Ant's standard echo task. That standard task is used for doing all that low level work, i.e. dumping text on streams on loggers. On top of it, some features have been implemented intended to generate nicely formatted output.

Here is the foobar example again:

```
| <c:echo>
```

⁶ [xmlint](#) is a good choice

⁷ Also known as *blank* character

```
foo\  
bar  
; supports continuation and \  
comment lines  
</c:echo>
```

This would output

```
[c:echo] foobar
```

which I believe is just what you had in mind.

Attributes

This task supports all attributes inherited from Ant's echo task. In addition, further supported attributes are:

Attribute	Type	Default	Description
<code>debug</code>	boolean	false	Enables additional debug output for this particular task.
<code>comment</code>	string	;	Allows for comments.
<code>shift</code>	string	``	Allows to prefix each line with <code>shift</code> characters. See also Behaviour below.

Notice that **debug** output will be written on stream `stderr` regardless whether `debug` has been globally enabled on Ant or not. Also standard Ant loggers and listeners are ignored. The default value is `false`, i.e. no additional output is created.

The trimmed `comment` attribute value is used to construct a regular expression like `^\s*\Q<<comment>>\E`. Every line matching this regular expression will not show up in the output. Notice that the comment value given does not allow for regular expression meta characters. Thus something like `(;|#)` does *not* mean either `;` or `#`. Instead it means that a line starting with `(;#)` is ignored from output. By default, lines starting with character `;` - like in Lisp - are ignored.

Elements

This task optionally accepts implicit text. That text may contain Ant property `${..}` or `EL #{..}` references.

Behaviour

Continuation Lines are lines where the last character before the line termination character is the backslash character. Such a line is continued, i.e. the line will be merged with the next one (which could also be a continuation line).

A (merge continuation) line starting with an arbitrary number of whitespace characters followed by the characters given in attribute `comment` is a **comment line**. Such lines are removed from output. The characters given are taken literally and have no meta character functionality. To disable comment lines altogether use an empty string ⁸.

To allow a **decent formatting** unnecessary whitespace characters are removed. The process is illustrated ⁹ using the introduction example used above:

```
<c:echo>
  ..foo\
  ..bar
</c:echo>
```

In a first step is the first non-whitespace character determined. In the example above, this is character `f`. From there Flaka counts backwards until a line termination character or the begin of input is reached. The counted number is the amount of whitespace characters stripped from the begin of each line. If a line starts with less than that amount of whitespace characters, then only those available are removed. Additionally, all whitespace characters before the first non-whitespace character are removed from the input.

There are two whitespace characters before `f oo\`. If support for continuation lines would have been disabled, Flaka would dump the following:

```
| [c:echo] foo\
```

⁸ A string consisting only of whitespace characters

⁹ Again character dot `.` is used to illustrate a whitespace character with the exception of line ending characters

```
| [c:echo] bar
```

Handling of continuation lines takes place **after** whitespace has been stripped.
Thus Flaka prints

```
| [c:echo] foobar
```

Flaka 1.01
häfelinger IT

as shown in the introduction example. A slight variation of the example above
is given next:

41/99

```
| <c:echo>
|
| ..foo\
| .bar
| ...indented by one character, right?
| </c:echo>
```

Notice that in front of `bar` is only one whitespace character while there are
three in the line after. What will be Flakas output?

```
| [c:echo] foobar
| [c:echo] .indented by one character, right?
```

As you can see, no more than the initial counted amount of whitespace is
removed from each line.

However, assume that you really want to have a couple of empty lines dumped
before any real content. How can this be done. There are two options. Firstly
you can always fall back to use Ants standard `echo` task. Secondly, you can
use a comment line like shown next

```
| <c:echo>
| ..; two empty lines following
|
| ..foobar
| </c:echo>
```

which would dump:

```
[c:echo]
[c:echo]
[c:echo] foobar
```

This all works because comment lines are removed from the input **after** the position of the first non-whitespace character gets determined. It obviously means that this kind of comments do matter and can't simply be stripped off. They may carry some semantics, so it's probably best to avoid this kind of trick. Make use of it when appropriate.

Flaka 1.01
häfelinger IT

42/99

We have seen how to force leading empty lines in the example above. What needs to be done if some leading whitespace is intended? Again there are two options. First you may attack the problem using the comment line trick:

```
<c:echo>
..; dummy comment
.....foobar
</c:echo>
```

This would produce like `[c:echo]foobar`. Or you may use the **shift** attribute to right-shift the whole output by an arbitrary amount of characters like

```
<c:echo shift="5">
..foobar
</c:echo>
```

producing the same as before, namely

```
[c:echo] .....foobar
```

Attribute `shift` expects a unsigned integral number followed by an optional arbitrary sequence of characters. This allows for a different *shift* character sequence as show next:

```
<c:echo shift="5">>
..foobar
</c:echo>
```

This produces >>>>> as shift character sequence for every line dumped as shown next:

```
| [c:echo] >>>>>foobar
```

Notice that every character after the integral number counts. Thus `5>` would produce

```
| [c:echo] > > > > > foobar
```

Flaka 1.01
häfelinger IT

43/99

instead.

This feature also allows to create some horizontal lines which might be useful to get attention for a particular message of importance like

```
| [c:echo] %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Those line of 40 per cent character % got created simply by using

```
| <c:echo shift="40%"/>
```

Further Links

- [Javadoc](#)
- [Source](#)

fail

This task has been derived from [Ants standard fail task](#). All attributes and elements are supported. When defining a message however, EL references can be used:

```
| <c:fail message="illegal state #{whichstate} seen" />
```

Flaka 1.01
häfelinger IT

44/99

Furthermore, attribute `test` has been added. The value of `test` will be evaluated as EL expression in a boolean context. Being `true`, `fail` will throw a build exception. When used in this way, `<c:fail test='expr' />` behaves exactly the same as

```
| <c:when test="expr">  
  <fail />  
</c:when>
```

Further Links

- [Javadoc](#)
- [Source](#)

for

A task that implements a loop statement. Iterating over a list of *objects*:

```
<c:for var="x" in=" list('a', 2, 'src'.tofile, typeof
  (list())) ">
  <c:echo>
    #{x}
  </c:echo>
</c:for>
```

Flaka 1.01
häfelinger IT

45/99

Attributes

Attribute	Type	Default	EL	Description
var	string		#{}	The variable holding each loop item. This variable can be referenced within the body like <code>#{var}</code> where <code>var</code> is the string used in this attribute. If not used, then no iteration takes place and no warning is issued. Notice that you can use <code>#{.}</code> only in EL enabled tasks.
in	string		expr	The items to be iterated over as EL expression. A iteration takes place except if <code>null</code> is the evaluation result. Otherwise, if the evaluation result is <i>not iterable object</i> , a temporary list containing that object is created on the fly. Iteration takes then place over that list and otherwise over the iterable collection.

Elements

The body of this task may contain an arbitrary number of tasks or macros.

Behaviour

This is the shortest possible for statement. Its legal albeit completely useless.

```
| <c:for />
```

Further Links

- [Javadoc](#)
- [Source](#)
- Task [for](#)
- Task [break](#)
- Task [continue](#)
- See section [Looping](#) for an introduction to looping in Flaka

Flaka 1.01
häfelinger IT

46/99

let

XML is not particular easy to read for humans. When assigning a couple of variables and properties, this becomes obvious. This elementary task allows to set multiple variables and properties in one go. In addition, comments and continuation lines are allowed for additional readability and comfort. For example:

Flaka 1.01
häfelinger IT

47/99

```
<c:let>
  f = 'folder'
  ; turn f into a file object
  f = f.tofile
  b = f.isdir ? true : false
  ; assign a *property*
  p := 'hello world'
  ; override a property if you dare
  p ::= "HELLO \
WORLD"
</c:let>
```

In this example, `f` is first assigned to be string "folder". The comment line - the one starting with character `;` - tells what the next line is going to do: turn `f` into a file object which can then be used further. Here we assign a variable `b` which becomes true if `f` is a directory.

While character `=` is used to assign a variable, use character sequence `:=` to assign a property instead. If such a property already exists, it will not be changed in accordance with Ants standard behaviour. If you dare and insist to override a property, use `::=` to do so.

Notice that the right side of `=`, `:=` and `::=` are in any cases a EL expression while the left side are expected to contain valid identifiers for variables and properties.

Attributes

Attribute	Type	Default	EL	Meaning
comment	string	;	no	The comment character sequence.

debug	bool	false	no	Turn on extra debug information.
-------	------	-------	----	----------------------------------

All attributes follow the rule that leading and trailing whitespace is ignored. Any attribute combination is allowed and will not result necessarily in a build error. If in doubt, turn on extra debug information.

Flaka 1.01
häfelinger IT

48/99

Elements

This task accepts implicit text. Text may contain any amount of [EL](#) and property references references. Continuation and comment lines are supported.

Behaviour

The comment character sequence is ";" by default. It can be changed to an arbitrary sequence using attribute `comment`. Once set, it cant be changed during the execution of this task. A comment characters are used to identify lines to be ignored from execution. Such a line is given if the first non whitespace characters of that line are identical with the sequence of comment characters. In other words, a line is being ingnored if matches the regular expression `^\s*<comment>`. The comment characters itself are not interpreted as regular expression characters. Therefore a given comment sequence like "(#|;)" does not mean that either ";" or "#" start a comment. Instead it means that a comment line starts with the characters "(#|;)" which would be rather awkward (while perfectly *legal*).

To support readability continuation lines are supported. Such a line is indicated by having \ as last character. Be careful not to put any whitespace characters after \, otherwise the line will not be recognized as such. Continuation lines are also working on comments as the example above shows. If a line is a continuation line, the last character \ is removed, the line is accumulated and the next line is read. If finally a non-continuation line is red (and only then), an evaluation of the accumulated line takes place: If the accumulated line is a comment it will be ignored and otherwise either treated as property or variable assignment.

Leading and trailing whitespace characters ignored in every (accumulated) line. For example, the property assignment `x := 'foo bar'` will assign the string

foo bar to property x. Notice that whitespace before and after x and before and *after* 'foo bar' is ignored. This is slightly different from reading Java properties where whitespace after 'foo bar' would *not* have been ignored!

When evaluating, each line is independent of other lines evaluated. Each line is evaluated in the order written. Evaluating means that the right side of the assignment is evaluated as EL expression and the resulting object is assigned to the variable stated on the left side. When evaluating properties, then the right side is evaluated into an object and additionally streamed into a sequence of characters (string).

Flaka 1.01
häfelinger IT

49/99

Notice that it is perfectly legal to use property or variable references as the following example shows:

```
<c:let>
  f = '${ant.file}'
  F = '#{f}'
</c:let>
```

Be aware that property references are evaluated *before* EL expressions. Consider:

```
<c:let>
  ;; let s hold string ant.file
  s = 'ant.file'
  ;; bad, f will not assigned
  f = ${#{s}}
</c:let>
```

The second assignment will not work as expected because, in a first step, all occurrences of `${..}` are resolved by Ant itself. In a second step, the expression `${#{s}}` will be evaluated. Since this expression is invalid, `f` will not be assigned.

Each line is evaluated in order. Therefore the following works as expected:

```
<c:let>
  s := '3 * 5'
  ;; defines r as 15
  r = ${s}
</c:let>
```

The following kind of meta programming will not work for `let`:

```
<c:let>
  property_or_var := condition ? '=' : ':='
  name ${property_or_var} expr
</c:let>
```

Flaka 1.01
häfelinger IT

50/99

In a first step all continuation lines are accumulated. Then each line is split in left and right part and in addition the assignment type. After that, properties are resolved on both sides by Ants property resolver. In an additional step are *EL references* evaluated on both sides. Eventually, the right side is evaluated as EL expression and its result is assigned to the stringized and whitespace-chopped left side.

Then meaning of `null` and `void`

Task `let` can also be used to *remove* variables and even properties. To illustrate this, here are example behaviours:

```
<c:let>
  x = 3 * 5
  ;; remove x
  x =
  ;; remove x
  x = null

  ;; let property p to '3*5' (a string)
  p := 3 * 5
  ;; ignored
  p := null
  ;; remove property 'p'
  p ::= null
  ;; .. same as
  p ::=
</c:let>
```

The following table gives an overview of the meaning of `null` and `void` ¹⁰ on the right side of an assignment:

¹⁰ `void` means that the absence of any characters

Assignment	Right Side	Result
=	null	If the right side evaluates to <code>null</code> , then the variable will be removed if existing.
=	<i>void</i>	The evaluation of an empty expression is <code>null</code> . See above how <code>null</code> is handled`
:=	null	Cause a <i>read only</i> property cant be removed, nothing will happen with this assignment. The property will also not be created.
:=	<i>void</i>	Same as := <code>null</code>
::=	null	Removes the property denoted by the left side
::=	<i>void</i>	Same as ::= <code>null</code>

Flaka 1.01
häfelinger IT

51/99

Further Links

- [Javadoc](#)
- [Source](#)

list

A elementary task to create a variable containing a *list* of objects.

```
<c:list var="mylist">  
  ;; each line is a EL expression  
  3 * 5  
  ;; each line defines a list element  
  list('a',1,').tofile)  
</c:list>
```

Flaka 1.01
häfelinger IT

52/99

Attributes

Attribute	Type	Default	EL	Meaning
var	string		r	The name of the variable to be assigned.
comment	string	;		The comment character
debug	bool	false		Turn on extra debug information.
el	bool	true	no	Enable evaluation as EL expression

Elements

This task may contain a implicit text element.

Behaviour

This task creates and assigns in any case a (possible) empty list, especially if no text element is present. The variables name is given by attribute `var`. This attribute may contain references to EL expressions.

If given text element is parsed on a line by line basis, honouring comments and continuation lines. Each line will be evaluated as EL expression after having resolved `#{. .}` and `#{. .}` references. A illegal EL expression will be discarded while the evaluation of lines continues. Turn on extra debug information in case of problems.

The evaluation of a valid EL expression results in an object. Each such object will be added to a list in the order imposed by the lines.

Flaka 1.01
häfelinger IT

A single line cant have more than one EL expressions. Thus the following example is invalid:

53/99

```
<c:list var="mylist">
  ;; not working
  3 * 5 'hello, world'
</c:list>
```

Use attribute `el` to disable the interpretation of a line as EL expression:

```
<c:list var="mystrings" el="false">
  3 * 5
  ;; assume that variable message has (string) value '
  world'
  hello, #{message}
</c:list>
```

This creates a list variable `mystrings` containing two elements. The first element will be string `3 * 5` and the second element will be string `hello, world`. Notice that even if EL evaluation has been turned off, EL references can still be used.

Further Links

- [Javadoc](#)
- [Source](#)

properties

A task to set multiple properties in one go. It is typically used to *inline* properties otherwise written in an additional properties file. Thus using this task reduces the clutter on your top level directory:

```
<c:properties>
  ; this is \
  a comment

  ; assume that variable 'foo' has been defined here
  and that
  ; foo.name resolves into 'foo', then the next line
  will set
  ; property foo to be the string 'foo'.
  foo      = #{foo.name}
  ; next lines creates property 'foobar' to be the
  string 'foobar'.
  foobar   = ${name}bar
</c:properties>
```

Flaka 1.01
häfelinger IT

54/99

Attributes

Attribute	Type	Default	EL	Description
debug	boolean	false	no	Turn extra debug information on
comment	String	;	no	The character that starts a comment line

Elements

This task accepts a implicit text element.

Behaviour

This task is similar to [let](#). The difference is that this task only allows to define properties while [let](#) also supports the creation of variables. Furthermore, the right side of = will be literally taken as string value. This is different from [let](#) where the right side will be additionally evaluted as [EL](#) expression. The

following example defines each property `foobar`, once done with task `let` and once with this `properties` task:

```
<c:let>
  foobar := 'foobar'
</c:let>
<c:properties>
  foobar = foobar
</c:properties>
```

Flaka 1.01
häfelinger IT

55/99

Notice the usage of the quote character `'` in the former example and the absence of it in the latter.

Task `properties` supports, like task `let` does, continuation lines and comments. Furthermore, variable references `#{..}` and property references `${..}` are resolved on both sides of `=`.

If the right side is empty, then no property will be created and an existing property will not be changed. If the right side is `null`, a property with string value `null` will be assigned if the property does not already exist (this is very much different than when using task `let` to create properties).

Leading and trailing (!) whitespace characters are ignored. This is different from standard Ant where trailing whitespace is significant (and responsible for unexpected and hard to track script behaviour).

Further Links

- [Javadoc](#)
- [Source](#)

rescue

Task `rescue` is essentially a container for an arbitrary number of tasks. In addition, it allows to rescue variables and properties.

```
<c:rescue>
  <vars>
    foo
  </vars>
  <properties>
    bar
  </properties>
  task_1
  ..
  task_N
</c:rescue>
```

Flaka 1.01
häfelinger IT

56/99

No matter what will happen with property `var` and variable `foo` within `sequential`, this will be unnoticeable outside of `rescue` cause the values (or better: state) will be restored after having executed all embedded tasks. This will of course also work in case an exception is thrown by one of the tasks.

Attributes

This task does not define attributes.

Elements

Name	Cardinality	Description
<code>vars</code>	0..1	Defines a list of variable names. Attributes and behaviour is that of task list except that interpretation of lines as EL expressions are disabled.
<code>properties</code>	0..1	Defines a list of property names. Attributes and behaviour is that of task list except that interpretation of lines as EL expressions are disabled.

<i>task</i>	arbitrary	A (arbitrary) task or macro to be executed
-------------	-----------	--

Behaviour

Executes all embedded tasks. Variables and properties listed in `vars` and `properties` are restored to their previous state, i.e. if not existing before the execution, they will not exist afterwards. If existed, their value will be restored.

Flaka 1.01
häfelinger IT

57/99

Further Links

- [Javadoc](#)
- [Source](#)

switch

Task `switch` has been designed to provide for simple pattern matching. The idea is to match a series of patterns - either [regular](#) or a [glob](#) expressions - against a *string value* and carry out a sequence of actions in case of a hit. Here is an illustrative example:

```
<c:switch value=" a${string}#{value} ">
  <matches glob="*.jar">           -- #1
    -- string ending in .jar
  </matches>
  <matches re="1|2|3">           -- #2
    -- one or two or three
  </matches>
  <matches re="-\d+">           -- #3
    -- negative integral number
  </matches>
  <otherwise>
    -- no match so far ..
  </otherwise>
</c:switch>
```

Flaka 1.01
häfelinger IT

58/99

Notice the usage of a glob expression in the first and the usage of regular expressions in the second and third `matches` element. Utilization of glob and regular expressions make `switch` a very flexible and powerful conditional statement.

Attributes

Attribute	Type	Default	Description
value	string	""	The string value that needs to be matched against.
var	string	-	Save details of this match as <code>matching object</code> using the variable name given.
ignore-case	bool	false	Enables case-insensitive matching.
comments	bool	false	Permits whitespace and comments in pattern.
dotall	bool	false	In <code>dotall</code> mode, the literal <code>.</code> matches any character, including a line terminator.

<code>unixlines</code>	bool	false	In this mode, only character LF is accepted as line terminator character when using <code>.</code> , <code>^</code> , and <code>\$</code> .
<code>multiline</code>	bool	false	In multiline mode, the literals <code>^</code> and <code>\$</code> match just after or just before, respectively, a line terminator or the end of the input sequence.
<code>debug</code>	bool	false	Turn on extra debug information
<code>matches.re</code>	string		Element <code>matches</code> : Specify a matching pattern as regular expression.
<code>matches.pat</code>	string		Element <code>matches</code> : Specify a matching pattern as glob expression

Flaka 1.01
häfelinger IT

59/99

Note that switch's `*value(given` is *normalized*. Leading and trailing whitespace is removed. This attribute is [EL](#) enabled but notice that you have to use EL references like `a#{ expr }b`.

A matcher object is saved in variable **var**. This matcher object allows to access matching details like the number of capturing groups and the like. See [matcher properties](#) for a list of available properties; see also below for examples. The variable name must be a string which may contain EL references.

By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case-insensitive matching can be enabled by specifying the `UNICODE_CASE` flag in conjunction with this flag. Case-insensitive matching can also be enabled via the embedded flag expression `(?i)`.

In this mode, whitespace is ignored, and embedded comments starting with `#` are ignored until the end of a line. Comments mode can also be enabled via the embedded flag expression `(?x)`.

By default `.` (the dot) match any character but line terminators. This can be changed by setting the **dotall** attribute to `true`. Dotall mode can also be enabled via the embedded flag expression `(?s)`, where `s` is a mnemonic for *single-line* mode, which is what this mode is called in [Perl](#).

Unix lines mode can also be enabled via the embedded flag expression `(?d)`.

By default `^` and `$` only match at the beginning and the end of the entire

input sequence. By setting **multiline** to true do they match just after and just before a line termination character. Multiline mode can also be enabled via the embedded flag expression (?m).

Note that each of the above `switch` attributes can also be applied to a `matches` element. Applied on `switch` has the effect of providing the default value for subsequent `matches` elements.

Flaka 1.01
häfelinger IT

60/99

Elements

Element	Cardinality	Description
<code>matches</code>	0..infinity	An element to specify pattern.
<code>default</code>	0..1	The default statement will be executed if no <code>matches</code> element matched the input value.
<code>otherwise</code>	0..1	This element is a synonym for element <code>default</code>

Element **matches** supports all the attributes of the enclosing `switch` with the exception of `value`. It may contain any number of tasks or macros as sub elements. They are carried out if the pattern matches the given switch value.

The **default** element is carried out if no pattern matched. This element is optional and can only be specified once. A build exception will be raised if used more than once. It does not accept any attributes. It may contain any number of tasks or macros as sub elements.

Behaviour

Attribute `value` is the basis for all further matching. It is a string value which may contain [property](#) or [EL](#) references.

Any number of `matches` elements are allowed and at most one `otherwise` or `default` element. Whether the `otherwise` element is at the end, in the middle or at the begin does not matter. The order of `matches` is relevant however. Each `matches` element is tried against the value in the order given. Then, if no element matched, a given `otherwise` or `default` element is carried out. Otherwise the winning matching element will while remaining elements are ignored.

Carrying out an element means that all enclosed tasks or macros are executed in the order given.

The underlying regular expression engine is the one given by Java. Its [Javadoc](#) documentation is a pretty good source of information if you are familiar with regular expressions. For all the gory details, have a look at [Mastering Regular Expressions](#) by Jeffrey E. F. Friedl.

Flaka 1.01
häfelinger IT

61/99

Be aware that there is no need to escape the escape character. For example, people using regular expressions in Java are used to write `*` if they want match the literal `*` character and thus escaping from the usual semantics (match zero or more times). This is not necessary in Flaka where the input sequence `*` remains `*`.

So called *globs* are a kind of simplified regular expressions. They lack the full power while simplifying the expression. For example, to specify whether a name input string end in `jar`, we can simply write

```
<c:switch value=" #{myfile}.name ">
  <matches glob="*.jar">
    -- do something with jar file ..
  </matches>
</c:switch>
```

The very same can also be expressed as `re=".jar$"` using regular expressions. The biggest disadvantage of globs are that capturing groups are not supported. Thus the match above just indicates that the file name ends in `.jar` while there is nothing to figure the files basename. Compare this with

```
<c:switch value=" #{myfile}.name ">
  <matches re="^(.*)\.jar$" var="m">
    <c:echo>
      basename = #{m[1]}
    </c:echo>
  </matches>
</c:switch>
```

Here we use a capturing group for the basename. A matcher object will be associated with variable `m`. This object can then [evaluated using properties](#) for matching details.

Here is a more complicated example. It was used once to examine a CVS tag which was supposed to follow the convention `schema-(env_)version`, where `(env_)` was optional, `schema` indicated the tags semantic and where `version` was the products version or build number:

```
<c:switch value=" 'v-uat_3_20_500' " var="m">
  <matches re="v-(?:([\d][^_]*_)?(\d.*))" >
    <c:echo>
      pattern      = #{m.p}          -- v-(?:([\d][^_]*_)?(\d.*))
      groups       = #{m.n}          -- 2
      matched text = #{m}            -- v-uat_3_20_500
                        (same as m[0])
      env          = #{m[1]}         -- uat
      version      = #{m[2]}         -- 3_20_500
      ;; referring to non existing group
      ??          = #{m[3]}         -- (empty string)
      ;; start and end index of first group
      start       = #{m[1].s}       -- 2
      end         = #{m[1].e}       -- 5
    </c:echo>
  </matches>
</c:switch>
```

Flaka 1.01
häfelinger IT

62/99

Further Links

- [Javadoc](#)
- [Source](#)

throw

A task to re-throw a previously thrown exception. If no exception has been thrown before, a new exception is thrown. In that case, throw acts like standard fail task .

Note that throw would re-throw the last thrown exception - regardless of the current context. The following would therefore work:

```
<c:trycatch>
  <try>
    <fail message="4711" />
  </try>
  <catch>
    -- handle the exception ..
  <catch>
</c:trycatch>
..
.. -- very much later
..
<c:throw /> -- re-throws "4711" exception!!!
```

Flaka 1.01
häfelinger IT

63/99

Attributes

Attribute	Type	Default	EL	Description
reference	string	trycatch-object	no	The name of the reference holding the previously thrown exception
var			no	Same as reference

Behaviour

A typical usage example:

```
<c:trycatch>
  <try> ..<fail message="4711"/> </try>
  <catch>
    <echo>caught exception ..</echo>
  <c:throw />
```

```
</c:catch>  
</c:trycatch>
```

When being executed, Ant would receive a build exception (re-thrown within the catch clause) containing "4711" and terminate.

Further Links

- [Javadoc](#)
- [Source](#)
- Task [trycatch](#)

Flaka 1.01
häfelinger IT

64/99

trycatch

A task mirroring try-catch-finally exception handling found in various languages.

All tasks inside try are executed in order. If an exception is thrown by one of them, then several things may happen:

- If there is a matching catch clause, then all tasks in that clause are executed. If there isn't a catch clause, the exception will be passed to the enclosing environment (except if an exception is also thrown in the finally clause - see below).
- An optional finally clause is always executed, regardless of whether an exception gets thrown or whether a try or catch clause exists.
- If a property is set, then that property will hold the message of the exception thrown in a try clause. If a reference is given, then that reference will hold the exception object thrown in the try clause. If an exception is also thrown in a catch or finally clause, then neither will the property or reference update nor set.
- If an exception is thrown in a matching catch clause and in a finally clause, then the latter will be passed to the enclosing environment and the former will be discarded.

A catch clause can be given a type and a match argument. Both arguments expect a regular or pattern expression. A catch clause matches if the type and match matches. The type argument is matched against the classname of the thrown exception. The match argument is matched against the exception message (if any). Both default values ensure that a build exception thrown by Ant is caught while an implementation dependent exception passes.

When matching against the error message, be aware that the actual error message might be slightly different from the actual message given: usually the error message contains also information about where the exception got thrown. It is therefore wise to accept any leading and trailing space. For example:

```
<c:trycatch>
  <try><fail message="4711" /></try>
  <catch match="4711">
    -- does (very likely) not match
```

```

</catch>
<catch match="4711*">
  -- neither this one ..
</catch>
<catch match="*4711">
  -- bon chance
</catch>
<catch match="*4711*">
  -- this is it!
</catch>
</c:trycatch>

```

Flaka 1.01
häfelinger IT

66/99

Attributes

Attribute	Type	Default	EL	Description
property	string		no	The name of the property that should hold the exception message
reference	string	trycatch-object	no	>The name of the reference to hold the exception object
catch-type	glob	*.BuildException	no	A pattern against the type (Java classname) of the exception object
catch-match	glob	*	no	A pattern to be applied against the exception message

Elements

try	A task container to hold tasks and macros to be given a try.
catch	A task container to be executed if an exception gets thrown
finally	A task container to be executed in any case

Note that all elements are optional. However, if there's no try element, then there's no chance to execute catch at all, so this constellation does not make

too much sense. The optional finally clause will be executed regardless of whether a try clause exists or not.

It is allowed to have more multiple try, catch or finally clauses and further does the order in which they appear not matter. Be aware though that eventually all try and finally clauses are merged into one try resp. finally clause.

Flaka 1.01
häfelinger IT

67/99

Behaviour

The following snippet demonstrates the usage of trycatch:

```
<c:trycatch property="reason">
  <try>
    <echo>1st try ..</echo>
  </try>
  <try>
    <echo>2nd try ..</echo>
    <fail message="fail within 2nd try" />
  </try>
  <try>
    <fail message="fail within 3rd try" />
  </try>
  <catch type="*.BuildException" match="*">
    <echo>..caught : ${reason}</echo>
  </catch>
  <finally>
    <echo>..finally</echo>
  </finally>
</c:trycatch>
```

Giving:

```
[echo] 1st try ..
[echo] 2nd try ..
[echo] ..caught : fail within 2nd try
[echo] ..finally
```

Further Links

- [Javadoc](#)
- [Source](#)

unless

This task is the logical opposite of task [when](#). Its body is only executed if the condition evaluates to false. See [when](#) for details. This example shows how to create a folder named libdir if such a folder does not already exist.

```
<c:unless test=" 'libdir'.tofile.isdir ">  
  <mkdir dir="libdir" />  
</c:unless>
```

Flaka 1.01
häfelinger IT

68/99

Further Links

- [Javadoc](#)
- [Source](#)

unset

The unset statement allows the removal of properties. Use this task with care as properties are not meant to be changed during execution of a project.

```
<c:unset>
  p1
  ;; use embedded EL references for dynamic names
  p#{ index }
</c:unset>
```

Flaka 1.01
häfeling IT

69/99

This example demonstrates how to remove properties `p1` and a property whose name depends on the current value of `index`.

Attributes

Attribute	Type	Default	EL	Description
debug	boolean	false	no	Turn extra debug information on
comment	String	;	no	The character that starts a comment line

Elements

This element accepts implicit text.

Behaviour

Each non comment line defines a property name to be removed. The property does not need to exist to be removed. User properties (i.e. given by command line) and system properties (i.e. `ant.file`) are also removed.

Comment lines and empty lines are ignored. Continuation lines, i.e. lines ending in `\` but not in `\\`, are accumulated before being processed.

References to properties `${..}` and expressions `#{..}` are resolved.

The content of a line defines the property name, for example:

```
<c:unset>
```

```
;; property 'foo bar', not 'foo' and 'bar'
foo bar

;; a line is *not* a EL expression (this will be
   property '3 * 5')
3 * 5

;; use #{..} references for dynamic content (this
   will be 'p15')
p#{3*5}
</c:unset>
```

Flaka 1.01
häfelinger IT

70/99

Further Links

- [Javadoc](#)
- [Source](#)

when

Task `when` represents a else-less if statement. The following example dumps the content of a file to stdout via Ants `echo` task if the file exists.

```
<c:when test=" 'path/to/file'.tofile.isfile" >
  <c:let var="fname" property="true" value=" f " />
  <loadfile property="__z__" srcFile="{fname}"/>
  <echo message="{__z__}" />
</c:when>
```

Flaka 1.01
häfelinger IT

71/99

Note that the example is bit artificial cause Ants `loadfile` task is sufficient.

Attributes

Attribute	Type	Default	EL	Description
<code>test</code>	string	false	expr	A EL expression that must evaluate to <code>true</code> in order to execute the body of this if statement.

Elements

- Any tasks or macro instances.

Further Links

- [Javadoc](#)
- [Source](#)

while

A task implementing a `while` loop:

```
<c:let>
  i = 3
</c:let>
<c:while test=" countdown >= 0 ">
  <c:echo>#{countdown > 0 ? countdown : 'bang!' }</c:
    echo>
</c:while>
```

Flaka 1.01
häfelinger IT

72/99

Attributes

Attribute	Type	Default	EL	Description
test	string	false	expr	The condition for looping as EL expression

Elements

The body of this task may contain an arbitrary number of tasks or macros.

Behaviour

All tasks listed as elements are executed as long as the [EL](#) expression evaluates to true.

Further Links

- [Javadoc](#)
- [Source](#)
- [break](#) to stop the iteration
- [continue](#) to hide tasks from being executed during a iteration step.
- See also section [Repetition](#) for an introduction to looping in Flaka.

Part III, Special Purpose

Flaka 1.01
häfelinger IT

73/99

export

Use this task to dump an arbitrary file from Flaka's jar to stdout or a file. This task's usefulness is rather limited for public use, however it allows you to have a look at Flaka's `antlib.xml`.

Flaka 1.01
häfelinger IT

Attributes

74/99

Attribute	Type	Default	EL	Description
<code>dst</code>	string	(stdout)	no	The destination to dump the file to. If <code>dst</code> is <code>-</code> the file will be dumped to stdout.
<code>src</code>	string	<code>antlib.xml</code>	no	The file within the package to export. By default, file <code>antlib.xml</code> is exported.
<code>tee</code>	bool	false	no	If enabled, the file will be exported to <code>src</code> as well as to stdout (similar as standard UNIX command <code>tee</code> does).

Further

- [Javadoc](#)
- [Source](#)

find

Ant lacks a simple task to report all or certain files in a folder. This task lets you evaluate an arbitrary [fileset](#) into a list of `file`s. Used in conjunction with [task for](#) enables you to print easily all files and properties of them. Of course, that generated list can also be used for other purposes.

```
<c:find var="filelist" type="f" />
<c:for var="file" in="filelist">
  <c:echo>
    #{file} has been last accessed at #{file.mtime}
  </c:echo>
</c:for>
```

Flaka 1.01
häfelinger IT

75/99

Attributes

Attribute	Type	EL	Default	Description
dir	string	=	Basedir of build script (".tofile)	The directories to scan.
type	string		all files and folders	f to select files and d to select folders
var	string	r	null	The name of the variable to hold a list of scanned files or folders

Elements

This task implements a [standard Ant fileset](#). All elements of `fileset` are therefore legal elements of `find` as well.

Behaviour

This task scans all files and folders given by attribute `dir` into a variable given by `var`. If `var` is empty, then no scanning takes place. Attribute `var` may

contain [EL references](#) which are resolved. If a scanning takes place, then `var` will be created and contains, in any case, a list value.

Attribute `dir` may contain [EL references](#) which are resolved. The so resolved string value must be a syntatically legal EL expression. This expression will then be evaluated. If the evaluated object is a list, then each list item will be scanned for files and folder, otherwise only the evaluated object. If such a object to be scanned is not already a file object, then stringized object is used to create the folder to be scanned. The following examples illustrate this behaviour:

Argument	Same as
<code>src</code>	<code>list(src.toFile)</code>
<code>list(src,lib)</code>	<code>list(src.toFile,lib.toFile)</code>

Attribute `type` is used to filter out unwanted files or folder from being scanned. If `type` is `f`, then only files are taken into account and when `d`, only folders matter. By default files and folders are accumulated.

Examples

Report all files in the working directory. Note that also files in sub folders are reported.

```
| <c:find var="filelist" type="f" />
```

Check whether a sub folder ending in `.jar` exists which is not a file but a directory.

```
| <c:find var="filelist" type="d">  
  <include name="**/*.jar" />  
</c:find>
```

Further

- [Javadoc](#)
- [Source](#)

run-macro

A task to invoke a macro or task (or a list of them) dynamically.

```
<macrodef name="foobar" >
  <sequential>...</sequential>
</macrodef>
..
..
<foobar />                                -- conventional way of
    using macro foobar
<c:run-macro name="foobar" />             -- dynamic way
```

Flaka 1.01
häfelinger IT

77/99

The current version does not support calling macros or tasks with arguments.

Attributes

Attribute	Type	Default	EL	Description
name	string		no	The name of the macro to run. Use whitespace chars to specify a list of names.
fail	bool	false	no	Whether a exception shall be thrown if macro does not exist

Further

- [Javadoc](#)
- [Source](#)

run-target

A task to invoke a target dynamically.

Attributes

Attribute	Type	Default	EL	Description
<code>`name`</code>	string		no	The name of the target to invoke.
<code>`fail`</code>	bool	false	no	Fail if target does not exist

Flaka 1.01
häfelinger IT

78/99

Example

```
<target name="foobar">
  ..
</target>
..
<c:run-target name="foobar" />
```

Further

- [Javadoc](#)
- [Source](#)

create-target

Use this task to create a dynamic target within the current project. Usually targets are declaratively written within a build file. The following target

```
<target name="foo" depends="bar" description="do foo"
  >
  <foo-task />
</target>
```

Flaka 1.01
häfelinger IT

79/99

can also be written as

```
<c:create-target name="foo" depends="bar" description
  ="do foo" task="foo-task" />
```

Note: The current version allows only the specification of one task or macro within the body of the target to be created (like foo-task above).

Attributes

Attribute	Type	Default	EL	Description
name	string		no	The name of the target to create. If override is false, an already existing target with that name is not created.
task	string		no	The task or macro to execute
description	string		no	The informal description of this target
depends	string		no	Targets to execute before this target
override	bool	false	no	Whether to override an existing target.

Further

- [Javadoc](#)
- [Source](#)

set-default-target

A task to define the projects default target.

```
<project xmlns:c="antlib:ant.epoline" default="this">
  <c:set-default-target name="foobar" override="true
    "/>
  ..
  <target name="foobar">
    ..
  </target>
</project>
```

Flaka 1.01
häfelinger IT

80/99

This example changes the default target to be called from this to foobar.

Attributes

Attribute	Type	Default	EL	Description
name	string		no	The name of the target
fail	bool	false	no	Whether to throw an exception if target does not exist
overri- de	bool	false	no	Whether to override an already existing default target

Further

- [Javadoc](#)
- [Source](#)

xmlmerge

A simple task to merge XML files.

Attributes

Attribute	Type	Default	EL	Description
dst	string	-	no	The file to write to. Any intermediate folders are created. writes to stdout if empty string or argument - is given.
src	string		no	A file or folder argument. If a folder is given, then any file therein matching a given pattern is merged, otherwise the file given. A relative argument is interpreted as being relative to the current working directory.
pattern	regex	.*\\- .xml	no	The pattern to apply when matching files to be merged. The pattern can be a regular expression
root		string	no	By default the root tag of the first element will be the root tag for the merged content. Use attribute root to specify a different root tag.

Flaka 1.01
häfelinger IT

81/99

Behaviour

Assume that folder xmldir contains two files x.xml and y.xml where x.xml looks like

```
<x>  
  <x>This is X/x</x>  
</x>
```

and where y.xml looks similar. Then the following invocation gets:

```
<c:xmlmerge dst = "-" src = "xmlmdir" pattern = "*.xml"
  " root = "myroot" />
<?xml version="1.0" encoding="UTF-8"?>
<myroot>
<X>
  <x>This is X/x</x>
</X>
<Y>
  <y>This is Y/y</y>
</Y>
</myroot>
```

Flaka 1.01
häfelinger IT

82/99

The same setup as before gives, when no root element is applied, the following:

```
<c:xmlmerge dst = "-" src = "xmlmdir" pattern = "*.xml"
  " />
<?xml version="1.0" encoding="UTF-8"?>
<X>
  <x>This is X/x</x>
  <Y>
    <y>This is Y/y</y>
  </Y>
</X>
```

Further

- [Javadoc](#)
- [Source](#)

property-by-regex

A task to compose a property based on existing properties.

The new property is composed by searching for property names matching a given regular expression. Then the new property is created by concatenating the value of each matching property using a separator of choice.

Flaka 1.01
häfelinger IT

83/99

Attributes

property	The name of the property to create
regex	The regular expression.
sep	The separator to use when concatenating

Example

Assume that the following properties are defined:

```
depot.1.url = http://depot/component  
depot.2.url = http://depot/3rdparty  
depot.3.url = http://depot/externals
```

Then

```
<c:property-by-regex property="depot.csv" regex="depot\\.\\d+\\.url" />
```

creates property

```
depot.csv = http://depot/component,http://depot/3rdparty,http://depot/externals
```

Further

- [Javadoc](#)
- [Source](#)

scandeps

A task to scanning for dependencies in files.

```
<c:scandeps var="mydeps">  
  <include name="build.xml" />  
</c:scandeps>
```

Flaka 1.01
häfelinger IT

84/99

Example above searches for dependencies in file `build.xml` in the [project's base folder](#). The list of dependencies is assigned with variable `mydeps`.

Attributes

Attribute	Type	Default	EL	Description
<code>var</code>	string		<code>#{} </code>	The name of the variable to hold dependencies
<code>dir</code>	string	<code>".tofile</code>	yes	The root folder to use when scanning files
<code>debug</code>	bool	false	no	Turn on extra debug information

Elements

This task is a implicit [fileset](#) and may thus contain any elements which can be applied to a fileset.

Behaviour

If attribute `var` is not used, then no scanning for dependencies takes place and no warning will be issued (except when `debug` has been enabled). If a variable is used, then a list of dependencies will be assigned in any case. If no dependencies are found, the list will be empty.

Attribute `dir` is evaluated as [EL](#) expression. If the result is a file object, the object is taken as the root for scanning files. If `dir` is not used, then the current [basefolder](#) is used as root.

This task may take any elements which are applicable for a [fileset](#) type. If no elements are used, then all files and folders beneath `dir` are scanned. Otherwise

the elements applied will limit the scan for specific files or folders.

A dependency is just recognized by its element name regardless any namespace.
Therefore something like

```
<dependency />  
<x:dependency>
```

Flaka 1.01
häfelinger IT

will be accepted as dependency when scanning.

85/99

Further

- [Javadoc](#)
- [Source](#)

install-property-handler

A task to install Flakas property handler. When installed, Ant *understands* EL references like `#{. .}` in addition to standard property references `${. .}`.

An example will illustrate this:

```
<c:let>
  ;; let variable foo to string 'bar'
  foo = 'bar'
<c:let>
<echo>
  [1] #{foo}
</echo>
<c:install-reference-handler />
<echo>
  [2] #{foo}
</echo>
```

Flaka 1.01
häfelinger IT

86/99

Assume in this example, that the standard Ant property handler is installed. In the first `<c:let/>` task you can use EL because this task is provided by Flaka and thus EL aware. This is not the case for the `<echo/>` task following. Thus something like `#{foo}` has no meaning. However, after Flakas property handler is installed, the situation changed.

This is the output of aboves snippet:

```
[echo] [1] #{foo}
[echo] [2] bar
```

Attributes

Attributes	Type	Default	EL	Description
type	string	elonly	<code>#{}</code>	Install handler with certain additional features enabled (see below)

Behaviour

If `type` is `only` (exactly as written), then the new handler will only handle `#{..}` in addition. If `type` is `remove`, then unresolved property references are discarded.

Further Links

- [Javadoc](#)
- [Source](#)

Flaka 1.01
häfelinger IT

87/99

logo

A small handy task to create a kind of *framed* text like

```
.....  
::          Text          ::  
::      more text      ::  
.....
```

Flaka 1.01
häfelinger IT

88/99

This kind of *logo* is easily created and dumped on standard output stream like

```
<c:logo chr="::" width="20">  
  ;; here is my text  
  Text  
  more text  
</c:logo>
```

where `chr` defines the *wrap character* - here `::` and `width` defines the overall length of one line - here 20 characters.

Attributes

Attributes	Type	Default	Description
<code>chr</code>	string	<code>:</code>	The wrapping character ..
<code>width</code>	integer	80	Width of one line in terms of regular characters.

Elements

This task accepts an implicit text field. This text field may contain comments (`;`) and leading whitespace is ignored. Thus the same rules as for task [echo](#) are applying here.

Behaviour

Contents of text element is read line by line. Comment lines are ignored. Leading whitespace is ignored. Leading whitespace is determined by the first non-whitespace character. See also task [echo](#) for details.

The contents of each line is centered. Leading and trailing space is filled up with *blanks_* in order to reach the given line width. If a line is longer than width, then all contents after width characters is silently skipped.

Further Links

- [Javadoc](#)
- [Source](#)

Flaka 1.01
häfelinger IT

89/99

scandep

A task for scanning dependencies in arbitrary XML files. Each dependency is converted into a dependency object which can then be accessed via EL properties. Here is an example:

```
<c:scandep var=" dependencies ">
  <decorate>
    ;; local storage
    dest = 'build/lib'
  </decorate>
  <include name="build.xml" />
</c:scandep>
```

Flaka 1.01
häfelinger IT

90/99

Then, assume that you have a *dependency* like

```
<c:dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
</c:dependency>
```

in *build.xml*. Construct `<c:dependency />` is provided by Flaka. It is implemented as macro and does nothing. Its only purpose is to allow an author to express an artifact via a Maven like dependency structure.

After `<c:scandeps />` has been executed, variable `dependencies` exists in any case and can be used in EL expressions like

```
<c:echo>
  ;; dependencies.each contains just all
  ;; dependencies found.
  There are \
  #{ size(dependencies.each) } \
  in total.

  ;; contains all dependencies in
  ;; scope compile
  Listing dependencies in scope compile:
  #{ dependencies.scope.compile }
</c:echo>
```

How about fetching a dependency from a remote repository like `http://repo1.maven.org`? No problem at all using Ant's `get` task along with properties provided by a dependency object ¹¹:

```
<c:for var=" d " in=" dependencies.each ">
  <get url=" ${repo}#{ d.m2path } " dest="#{ d.file }"
    />
</c:for>
```

Flaka 1.01
häfelinger IT

91/99

This would download the `junit` artefact from `http://repo1.maven.org` into folder `build/lib` assuming that property `repo` would contain the URL of the Maven repository. The artefact ends up in folder `build/lib` because of the decoration argument given:

```
<decorate>
  dest = 'build/lib'
</decorate>
```

Here we set the destination folder where we would like to store downloaded dependencies. Notice that `<:scandeps />` does not download any dependencies. It merely creates dependency data objects. Most information carried with such a data object is retrieved from the dependency itself. Other information, like where to store locally, is not part of a dependency's declaration. Its subject to the build file. This is what the `decorate` element is good for. It can be used to set properties on objects. Notice that EL does not provide a way of changing object properties. Basically, EL is for reading only.

Attributes

Attributes	Type	Default	Description
<code>var</code>	string	<code>dependencies</code>	Create a <code>dependencies</code> object containing dependency objects in a certain structure.
<code>dir</code>	string	<code>\${base-dir}</code>	The base folder for scanning XML files

¹¹ The following example assumes that EL has been globally enabled, see task [property-handler](#) for details.

Elements

This task is an implicit fileset task and supports all those elements, especially `<include/>` and `<exclude />` elements. By default *files* are scanned for dependencies.

The following additional elements are supported:

Element	Description
decorate	<code>name = value</code> text field.

Flaka 1.01
häfelinger IT

92/99

The `decorate` element allows to set properties on each scanned dependency data object. Properties must be set in a `property-name = el-expr` fashion. Comments are allowed. The only property supported is property `dest`.

Data Objects

Task `<c:scandep />` creates a data object having the following properties:

Property	Type	Description
<code>each</code>	List	A list of dependency objects
<code>scope</code>	Object	A dictionary for scopes
<code>scope.compile</code>	List	A list of dependency objects in scope <code>compile</code>
<code>scope.scope</code>	List	A list of dependency objects in scope <code>scope</code>
<code>alias</code>	Object	A dictionary of aliased dependencies
<code>alias.alias</code>	Dependency	Dependency with alias <code>alias</code>

Each dependency object supports the following properties:

Property	Type	Description
<code>alias</code>	String	the unique name of a dependency (if any)
<code>alt</code>	String	alternative download location
<code>basename</code>	String	
<code>file</code>	File	<i>expected</i> local storage of artifact
<code>group</code>	String	<i>groupid</i> in Maven parlance
<code>location</code>	File	Location where dependency got scanned from
<code>m1path</code>	String	Maven 1 path
<code>m2path</code>	String	Maven 2 path
<code>name</code>	String	<code>_artifactId</code> in Maven parlance

rev	String	<i>version</i> in Maven parlance
scope	List	list of scopes (each being a string)
type	String	artifact's extension

When looking for element or attribute names, names are compared in a case-insensitive manner. Elements take precedence over attributes.

Flaka 1.01
häfelinger IT

Property **group** is extracted from attributes `groupid` or `group` or element `groupid`. Similar, property **name** is extracted from attributes `artifactid` or `name` or element `artifactid`. Eventually, property **rev** is taken from attribute or element `version` or attribute `rev`. Those rules ensure that regular **Maven** or **Ivy** dependencies are recognized as such. In case a dependency is a mix of Maven and Ivy attributes, then those from Ivy, i.e. `group`, `name` and `rev` win. However, please be aware that elements are considered stronger than attributes.

93/99

If no **type** attribute is given, then `jar` will be assumed. The regular **basename** is a composition of `name`, `rev` and `type`. If attribute or element `jar` is given, then property **basename** returns that value instead ¹²

A dependency can be in various **scopes**. Scopes can be used to classify dependencies, for example to create different classpath objects for compilation or testing. Property `scope` is extracted from attribute or element `scope`. The content is expected to be a whitespace separated list of scope names.

Recognition

Task `<c:scandeps />` uses a **duck typing** approach when looking for a dependency. Everything having dependency like elements or attributes is considered a dependency - regardless of other elements and attributes and *regardless of any namespace*.

The list of scanned dependencies is unique. The unique value of a dependency is calculated by property `m2path` - the Maven 2 like path.

Further Links

- [ScanDeps Javadoc](#)

¹² To support irregular base names.

- [Dependency Javadoc](#)
- [ScanDeps Source](#)
- [Dependency Source](#)

Flaka 1.01
häfelinger IT

94/99

Part IV, Installation

Download ¹³ latest version of Flaka and *drop ant-flaka.x.y.z.jar* into your local Ant installation. All that needs to be done is to put this jar into Javas classpath when running Ant. There are various techniques how to do so. Please read-on what exactly needs to be done.

¹³ see either <http://download.haefelinger.it/flaka> or use <http://code.google.com/p/flaka/downloads>

Ready, ..

The following **requirements** must be satisfied before you start:

- Flaka requires [Java 1.5](#) or newer. You can change the version by setting environment variable `JAVA_HOME`. Have also a look at [Ants Manual](#) for other environment variables to be used.
- [Ant](#) version 1.7.0 or newer.

Flaka 1.01
häfelinger IT

96/99

Charge, ..

The most primitive technique is to save `ant-flaka-x.y.z.jar` into Ants library folder `lib`. If you have no clue where Ant is installed, try

```
$ ant -diagnostics | grep ant.home  
ant.home: /opt/ant/1.7.1
```

Flaka 1.01
häfelinger IT

97/99

Saving something in Ants library folder may not work if you lack required permissions. There is also the disadvantage that when switching to another installation of Ant, you need to reinstall Flaka again. Therefore consider to use Ants standard option `-lib`:

```
$ ant -lib ant-flaka-x.y.z.jar
```

A pretty nice feature of option `-lib` is that if the argument is a folder, that folder is scanned for `jar` files. Therefore you may want to do something like this:

```
$ mkdir $HOME/lib/ant  
$ cp ant-flaka-x.y.z.jar $HOME/lib/ant  
$ ant -lib $HOME/lib/ant
```

This approach has the nice advantage that you simply can drop other `jar` files into folder `$HOME/lib/ant` to make them reachable without touching the original Ant installation. As already mentioned, this will get handy when you have either multiple Ant installations. Btw, notice that option `-lib` can be applied more than once if the jars reside in various folders. Finally notice that folders are not recursively searched.

When working from the command line it is rather annoying to provide option `-lib` for each and every call. Fortunatley, Ant recognizes environment variable `ANT_ARGS` which can be used to let `-lib` disappear:

```
$ ANT_ARGS="-lib $HOME/lib/ant"  
$ export ANT_ARGS
```

The drawback with this technique is that you need to make sure that this variable is set in every environment you start up Ant. This sounds perhaps

easier than actually done. Luckily again, Ant reads file `$HOME/.antrc` and `$HOME/.ant/ant.conf` on each and every startup. It is therefore recommended to set variable `ANT_ARGS` in one of this files ¹⁴ to make just every plain call to `ant` is aware of Flaka. For example:

```
$ cat $HOME/.antrc
ANT_ARGS="-lib $HOME/lib/ant"
$ ls $HOME/lib/ant
ant-flaka-x.y.z.jar
```

Flaka 1.01
häfelinger IT

98/99

¹⁴ when doing so, you dont need to *export* that variable

Fire!

To check whether your setup is proper, create a small build script and try to execute it. For example, try this

```
$ cat > build.xml << EOF
<project xmlns:c="antlib:it.haefelinger.flaka">
  <c:logo>
    Hello, #{property['ant.file'].tofile.name}
  </c:logo>
</project>
^D
```

Flaka 1.01
häfelinger IT

99/99

Then, when created, try to execute it with your Flaka equipped Ant. You should see something like

```
$ ant
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                               Hello, build.xml                               ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
BUILD SUCCESSFUL
[.]
```

if everything is well setup.

Colophon

This document got written in AsciiDoc markup and translated into DocBook by using the `asciidoc` command. From DocBook it got translated into \LaTeX using `dblatex` and from \LaTeX eventually into PDF by using \XeTeX .

Flaka 1.01
häfelinger IT

100/99