The flaka Manual

Wolfgang Häfelinger
häfelinger IT

February 18, 2010
version 1.0

Introduction

In the world of Java, build scripts are traditionally written in Ant and recently also in Maven.

When it comes to write a build script using Ant, it feels like using a Shell script in a rather awkward language (XML). Each Ant task solves a particular problem. This is similar to a Shell where you have this small masterpieces like `mkdir`, `cp`, `tar` plus some control structures to eventually being able to put the one big thing together.

flaka häfelinger IT

2/**??**

Writing a build script using a Shell is serious business. And so it is when using Ant. Ant does not provide you any abstraction how the project needs to be build. There is no underlying logic. In fact you, the author, need to know what to do. Step by step. Whats more, you have to use the unfriendly XML syntax and restrictions, a control structure is missing and you have to use immutable properties to communicate between tasks. Therefore, Ant scripts are large, notoriously difficult to understand, usually not portable (usuallyt they just work on the authors host) and each author uses a different set of targets and properties.

Maven on the other side provides a high abstraction of building a project. Instead of describing how the project needs to be build, just describe project details and reports you like to have and Maven figures 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 how a project needs to be build automatically. Thats actually not quite true. In fact, Maven only works fine when following conventions setup by the Maven team. When not en route, Maven gets difficult as well. But even when following conventions, the number of options in Maven are now endless and question the idear of a declarative approach. Have a look at Mavens POM being a never ending series of XML tags]. At the end, I found myself using Ant again.

Still Im not happy with Ant.

What Im missing is the full power of a programming language. Yes, I want to have conditionals, loops and exception handling. I want to have variables which I can set or remove for pleasure. Such variables can reference any kind of object not only strings. And I need a nice expression language to retrieve and calculate in a simple yet elegant way. And 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. This is what Flaka is about:

- Programming Tasks (conditional, loops, exception handling, ..)
- Embedded Expression Language (EL)
- Framework to do the *right* thing, yet allows to use standard Ant when necessary
- Dependency handling (legacy, to be replaced by Ivy)

This four pillars are Flakas approach to simplify the process of writing a build script with Ant. Notice that you are by no means forced to use all four pillars. You can for example just use the programming tasks with or without elements of EL while you dont need to get in touch with Flakas dependency handling instruments and neither with the framework.

The folling example of a complete build script shall demonstrate the idear how a build script using Flaka looks like:

```
<project xmlns:c="antlib:it.haefelinger.flaka">
 <c:build />
 <c:dependency alias="log4j" />
</project>
```

The author just lists the dependencies required to build the project. Flaka would do the rest by checking the underlying project structure:

- figure out what type of project should be build (jar, war, ear ..)
- figure out where projects source code, test cases etc are
- handle dependencies
- create targets like `clean`, `compile`, `package`, `test` automatically
- generate Javadoc and other reports

Current Status

Not all targets have been reached in the current version of Flaka (Release candidate 1). Programming tasks and EL are working fine and can be used. The other two pillars work partially but generally not recommended (yet) to be used.

Where to go from here?

- Download Flaka and read the installation page.
- Have a look at the basic scripting elements to get an overview of tasks, types and macros provided by Flaka. Have a closer look in the reference part of this manual for all the gory details of those tasks, types and macros.
- Make sure to look into the chapter about the expression language, it contains a lot of information on this enormous useful extension.
- Start writing build scripts using Flaka and give feedback.

Programming Constructs

This chapter provides an overview of programming constructs Flaka provides. This programming constructs are one of the Flakas pillars.

Strings

Like Ant, Flaka supports currently strings and, when applicable, pointer to resources (by referencing a symbol). Ant provides no functionality manipulate a string value and neither does Flaka. However, Flakas expression language contains string functions to create new strings.

Symbols

Symbols are names carrying associated data. The name of a symbol is a sequence of characters. The allowed characters are basically unlimited. It is recommended to stick with well known characters [a-zA-Z0-9._-]. Symbols can be used as variables, target, task, type or macro names.

- `<property name=sym value=expr />` Use sym as variable: assign the value of *expr* to *sym*. **A symbol associated with a string value is called a property**. Notice that Ant and Flaka provide further ways of creating properties.
- `<macrodef name=sym>` Use *sym* as macro name
- `<target namesym>` Use *sym* as target name
- `<taskdef name=sym>` Use *sym* as task name
- `<typedef name=sym>` Use *sym* as type name
- `id=sym` Use sym as reference: assign the evaluation of task (or macro) to *id*

Properties

To reference a property, enclose its symbol name with curly braces and prefix with the dollar character like:

```
<property name="x" value="99" />
<echo>
  value of property x is ${x}    -- .. is 99
</echo>
```

```
<property name="x" value="99" />
<property name="x" value="The quick brown fox .."/>
<echo>
  value of property x is ${x}    -- .. is 99
</echo>
```


It can be done using Flakas task let or unet as the following snippet demonstrates.

```
<property name="x" value="99" />
<c:let>
   x ::= "The quick brown fox .."
</c:let>
<echo>
  value of property x is ${x}    -- .. is The quick
     brown ..
</echo>
```

Properties have their own symbol table (as targets, tasks, macros and types have). This means for example that it is possible to have a property and a task *sharing* the same symbol name:

```
  <property name="foobar" ../>
  <macrodef name="foobar" ../>  -- property foobar not
      harmed!
```

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.

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.

when and unless

Task when evaluates an [?] expressio. 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 `unset` 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* evalutes 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 [?] 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.

Repetition

Flaka has a looping statement. Use task for to iterate over a *list* of items. Use break and [?] 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>
```

Attribute `in` will be evaluated as [?] expression. In the example above, that [?] 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):

```
<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, theres something seriously wrong and it does not make much sense to continue. Use attribute *type* to catch (runtime) exceptions thrown by the underlying implementation.

throw

Task throw throws a (build) exception.

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

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

Macros

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

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

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.

Part II

Each of the following chapters describes a task in all it's details.

break

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


```
<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 [Tasks#for for] or [Tasks#while 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

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/Break.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/Break.java Source]

choose

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

Attributes

|| Attribute || Type || Default || EL || Description || || *when*.test || string || `false` || = || A EL condition. When `true` corresponding clause will be executed. || || *unless*.text || string || `true` || = || A EL condition. When `false` corresponding clause will be executed. || || debug || boolean || `false` || = || Turn on extra debug information. ||

Elements

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

Behaviour===

Each `when` and `unless` clauses conditions are evaluated in order given until a claus 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>
```

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

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/Choose.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/Choose.java Source]

continue

A task mirroring a continue statement. When used within a [Tasks#for 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.


```
<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 [Tasks#for for] or [Tasks#while 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

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/flaka/Continue.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/flaka/Continue.java Source]
- Task [Tasks#for for]
- Task [Tasks#break break]

flaka
häfelinger IT
18/**??**

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 [1] 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* [2] character. If you execute this, you will get

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

This is definitly 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 intented to generate nicly formatted output.

Here is the `foobar` example again:

```
<c:echo>
```

[1] xmlint is a good choice
[2] 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 tasks supports all attributes inherited from Ants echo task. In addition, further supported attributes are:

Attribute	Type	Default	Description
`debug`	boolean	false	Enables additional debug output for this particular task.
`comment`	string	;	Allows for comments.
`shift`	string	``	Allows to prefix each line with `shift` 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 listeneres 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 [?] `#{..}` 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 literaly and have no meta character functionality. To disable comment lines altogether use an empty string [3].

To allow a **decent formatting** unnecessary whitespace characters are removed. The process is illustrated [4] 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 `foo\`. If support for continuation lines would have been disabled, Flaka would dump the following:

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

[3] A string consisting only of whitespace characters

[4] 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
```


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

```
<c:echo>

..foo\
.bar
...indended 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] .indended 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 cant simply be stripped off. They may carry some semantics, so its probably best to avoid this kind of trick. Make use of it when approbriate.

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 optinal 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
```

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 using

```
<c:echo shift="39%">
..%
</c:echo>
```

Further Links

- Javadoc
- Source

fail

This task has been derived from [http://ant.apache.org/manual/CoreTasks/-fail.html 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" />
```

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

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/Fail.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/Fail.java 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>
```


Attributes

|| Attribute || Type || Default || EL || Description || || `var` || string || || #{} || The variable holding each loop item. This variable can be referenced within the body like `#{var}` where `var` 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 `#{..}` only in [EL] enabled tasks. || || `in` || string || || expr || The items to be iterated over as [EL] expression. A iteration takes place except if `null` is the evaluation result. Otherwise, if the evaluation result is *not iterable object*, a temporary list containg 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

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/For.html Javadoc]

- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/For.java Source]
- Task [Tasks#for for]
- Task [Tasks#break break]
- Task [Tasks#continue continue]
- Quickref [BasicScriptingConstructs#Looping Looping] for an introduction to looping in Flaka

install-property-handler

A task to install Flakas property handler. When installed, Ant *understands* [?] 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>
```

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`	#{}	Install handler with certain additional features enabled (see below)

Behaviour

If `type` is `elonly` (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

let

XML is not particular easy to read for humans. When assigning a couple of varables 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:


```
<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	[?]	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.

Elements

This task accepts implicit text. Text may contain any amount of [?] 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 accumlated 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 [?] expression and the resulting object is assigned to the variable stated on the left side. When evaluating properties, then the right side is evaluted into an object and additionally streamed into a sequence of characters (string).

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* [?] 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>
```


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 additonal 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* [5] on the right side of an assignement:

[5] *void* means that the absense of any characters

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

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>
```


Attributes

Attribute	Type	Default	[?]	Meaning
var	string		r	The name of the variabled 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.

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

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

Use attribute `el` to disable the interpretation of a line as [?] 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 thateven 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>
```

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 [?] 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>
```


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 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>
```


No matter what will happen with property `var` and variable `foo` within `sequential`, this will be unoticable 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 || || `vars` || 0..1 || Defines a [#list list] of variable names. Attributes and behaviour is that of task [#list list] except that interpretation of lines as [EL] expressions are disabled. || || `properties` || 0..1 || Defines a [#list list] of property names. Attributes and behaviour is that of task [#list list] except that interpretation of lines as [EL] expressions are disabled. || || *task* || arbitrary || A (arbitrary) task or macro to be excecuted ||

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.

Further Links

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/Rescue.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/Rescue.java Source]

switch

Task `switch` has been designed to ease pattern matching. The idea is to try to match a pattern, a [http://en.wikipedia.org/wiki/Regular_expression regular expression] or [http://en.wikipedia.org/wiki/Glob_(programming) glob expression] against a given string value and carry out a sequence of actions in case of a hit.


```
<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:case>
```

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 || [EL] || Description || || `value` || string || - || el || The (string) value that needs to be matched against. Note that the value given is *normalized*, i.e. leading and trailing whitespace is removed. Whitespace characters are controlled by the underlying implementation which is Java in this case.|| || `var` || string || - || #{..} || Save details of this match as `matching object` using the variable name given. See [EL#Matcher_Properties matcher properties] for a list of available properties; see also below for examples. || || `ignorecase` || bool || false || no || Enables case-insensitive matching. 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)` || || `comments` || bool || false || no || Permits whitespace and comments in pattern. 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)` || || `dotall` || bool || false || no || In dotall mode, the literal `.` matches any character, including a line terminator. By default this expression does not match line terminators. 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 [http://en.wikipedia.org/wiki/Perl Perl]. || || `unixlines` || bool || false || no || In this mode, only character `LF` is accepted as line terminator character when using `.`, `^`, and `$`. Unix lines mode can also be enabled via the embedded flag expression (?d). || || `multiline` || bool || false || no || In multiline mode, the literals `^` and `$` match just after or just before, respectively, a line terminator or the end of the input sequence. By default these expressions only match at the beginning and the end of the entire input sequence. Multiline mode can also be enabled via the embedded flag expression (?m). || || `debug` || bool || false || no || Turn on extra debug information || || *matches*.`re` || string || || #{..} || Element `matches`: Specify a matching pattern as regular expression. || || *matches*.`pat` || string || || #{..} || Element `matches`: Specify a matching pattern as glob expression ||

Note that each `switch` attribute (but `value`) can be applied to a `matches` element. Applied on `switch` has the effect of providing the default value for subsequenct `matches` elements.

Elements

|| Element || Cardinality || Description || || `matches` || 0..infinity || An element to specify a single regular or a glob expression. This element supports all the attributes of the enclosing `switch` (but `value`). It may contain any number of tasks or macros as sub elements. They are carried out if the expression matches. || || `default` || 0..1 || The default statement will be executed if no `matches` element matched the input value. This element is optional. This element can only be specified once. A build exception will be raised if used more than once. This element does not accept any attributes. It may contain any number of tasks or macros as sub elements. They are carried out if no expression matched. || || `otherwise` || 0..1 || This element is a synonym for element `default` ||

Behaviour

Attribute `value` is the basis for all further matching. It is a string value which may contain [Glossary#Property_Reference references to properties] or [Glossary#Expression_Reference EL expressions]. Leading and trailing whitespace will be discarded after having resolved all references.

All other attributes (see above) are default values for subsequent `matches` elements. For example, setting attribute `debug` to `true` will turn on debug on all `matches` elements.

Any number of `matches` elements are allowed and at most one `otherwise` (resp. `default`) element. Whether the `otherwise` element is at the end, in the middle or at the begin does not matter. The order of the `matches` elements are relevant however. Each `matches` element will be tried in the order given until no more untried elements are left. Then, if no element matched, a given `otherwise` element is carried out. Otherwise the matching elment will.

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 [http://java.sun.com/-j2se/1.5.0/docs/api/java/util/regex/Pattern.html 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 [http://oreilly.com/catalog/-9780596528126/index.html Mastering Regular Expressions] by Jeffrey E. F. Friedl.

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 disadvange 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 be [http://code.google.com/p/-flaka/wiki/EL#Matcher_Properties 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 convetion `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
      ;; refering 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>
```

Further Links

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/flaka/Switch.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/flaka/Switch.java 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!!!
```

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

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/flaka/Throw.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/flaka/Throw.java Source]
- Task [Tasks#trycatch trycatch]

flaka
häfelinger IT
47/**??**

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 tehm, then several things may happen:

- If there is a matching catch clause, then all tasks in that clause are executed. If there isnt 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 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 a 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 clauses 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 accepted 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>
```


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 theres no try element, then theres 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 oder in which they appear not matter. Be aware though that eventually all try and finally clauses are merged into one try resp. finally clause.

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

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/Trycatch.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/Trycatch.java Source]

unless

This task is the logical opposite of task [Tasks#when when]. Its body is only executed if the condition evaluates to false. See [Tasks#when 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>
```

Further Links

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/Unless.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/Unless.java 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>
```


This example demonstrates how to remove properties `p1` and a property whose name dependes on the curent 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>
```


Further Links

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/Unset.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/Unset.java 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>
```


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

Attributes

|| Attribute || Type || Default || EL || Description || || `test` || string || `false` || expr || A [EL] expression that must evaluate to `true` in order to execute the body of this if statement. ||

Elements

- Any tasks or macro instances.

Further Links

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/When.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/When.java Source]

while

A task implemening a `while` loop:

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


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

- [http://javadoc.haefelingerit.net/net.haefelingerit.flaka/1.00/net/haefelingerit/-flaka/While.html Javadoc]
- [http://code.google.com/p/flaka/source/browse/trunk/src/net/haefelingerit/-flaka/While.java Source]
- [Tasks#break break] to stop the iteration
- [Tasks#continue continue] to hide tasks from being executed during a iteration step.
- See also [BasicScriptingConstructs#Looping Looping] for an introduction to looping in Flaka

Colophon

This document got written in Asciidoc markup and translated into DocBook by using the `asciidoc` command. From DocBook it got translated into LaTeXusing `dblatex` and from LaTeXeventually into PDF by using XeTeX.

flaka
häfelinger IT