The flaka Manual

Wolfgang Häfelinger
häfelinger IT

February 16, 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.

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 [Tasks#let] or [Task#unset] 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 [Tasks#when when] evaluates an [EL EL expr]. 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 [Tasks#unless 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 [Tasks#choose choose] tests each `when` condition in turn until an *expr* evalutes to `true`. It executes then the body of that when condition. Subsequent `when`s 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 [Tasks#switch 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.

Repetition

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


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 [EL#String_Properties properties of a string] and [EL#File_Properties 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, [Tasks#trycatch 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 [Tasks#throw 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.

EL, The Expression Language

The Java Unified Expression Language (JSR-245) is a special purpose programming (albeit not turing complete) 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 specification. To the contrary, the language can be made available in a variety of contexts.

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 (sort for Unified 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!
```

as output. Notice the usage of task [Tasks#echo echo]. When being tried with Ants standard echo task, a totally different output needs to be expected.

Most important, [#EL_References EL references] #{..} are not resolved but rather print as given.

Another EL Example

The code snippet following shows *EL* in action. The idea is to list all unreadable files in a certain directory (here the root folder). The snippet shows how EL is used in [#EL_Ready_Tasks Flaka various EL enabled tasks].

flaka
häfelinger IT

13/??

```
<c:let>
  root = '/'.tofile
  list = list()
</c:let>

<c:for var="file" in=" root.list ">
  <c:when test=" file.isdir and not file.isread ">
    <c:let>
      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"), this gives:

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

Disabling EL

By default, *EL* is enabled. *EL* can be disabled by setting property `ant.el` to `false` (exactly as written). For example:

```
<!-- globally disable EL --->
```

```
<property name="ant.el" value="false" />
```

If the property is not set, or set to a different value, then *EL* is enabled.

EL Ready Tasks

EL expressions can only be used in tasks which are *EL* ready. This are:

- [Tasks#let let]
- [Tasks#properties properties]
- [Tasks#when when], [Tasks#unless unless]
- [Tasks#for for]
- [Tasks#echo echo]

Further tasks to follow. See also how to enable EL on a [#Globally_Enabling_EL global level].

Globally Enabling EL

To enable handling of EL references on a global level - i.e. on all tasks, types or macros and independent of the vendor - use task [Tasks#install-reference-handler install-reference-handler].

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 [1].

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, `#{..}` is an EL expression.

Not so in this documentation. Here only the inner part, denoted by `..` is a *EL expression* while `#{ ..}` is considered a *reference to an EL expression*. A

[1] 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

reference to an expression is used in contexts which are partially evaluated. Take task [Tasks#echo 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 `#{..}` is good for:

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

In other contexts, like in `<c:when test=" condition " />`, a EL expression is expected anyway and it does not make the slightest sense to require the expression to be referenced. As an example, assume that we want to check whether a property named *foobar* exists. Instead of writing

```
<c:when test=" #{has.property['foobar']} " />    -- don
    't!
```

as seen in popular JSP tag libraries, just write

```
<c:when test=" has.property['foobar'] " />    -- yes!!!
```

And forget about that unnecessary clutter.

Notice however, that in all contexts where a expression is expected, a expression reference can be used. This allows for advanced meta programming like shown in the following example:

```
<c:when test=" has.property['#{propertyname}'] " />
    -- sic!
```

Handling of ${..}

EL defines two types of references: * **deferred**, indicated by #{..} ; and * **dynamic**, indicated by ${..}

Dynamic references ${..} are handled by Ant to resolve properties. There are two execptions to this however. Ant will leave a dynamic reference as is if the reference value does not denote a (existing) property. Secondly, Ant allows to escape a reference by by doubling character $ as in $${a}. In any case, ${..} does not denote a legal EL reference and will be left as is (notice that you can install a property handler to get rif of unresolved ${..} property references.

Handling of #{..}

Deferred references #{..} are evaluated according to regular EL rules. Each reference is evaluated independently. Thus

```
The #{ 'Good' }, the Bad and the #{ 'Ug' 'ly' }, a
    well known #{ 'movie' }.
```

Would print

```
The Good, the Bad and the , a well known movie.
```

cause the second reference is illegal. Notice however that all valid references are evaluated.

Nested References

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

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

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}.

Gory EL Details

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
```


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 buildin operators, [#Operators see below for details].

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

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` cant 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 associcated 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* quoation 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 quoation 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 cant 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 quoation character. Then, escape any occurences of the quoation and escape character within the string to preserve the original input string.

Here are same 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

Four *operators* are 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 [#Properties Properties] below.

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 evaluted, 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 practial 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 dont seem of any good use. The picture changes perhaps with this example:

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


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 [#Functions top level functions (see below)].

The following implicit objects are defined by Flaka: || Implicit Object || Type || Description || || *name* || || If *name* is not a predefined name as listed in the rest of this table, then *name* will be the same as `var[name]`, i.e. *name* will resolve to the object associated with variable *name*. || || project || || 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. || || property || || 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 constant e, also known as Euler's number. || || pi || double || The mathematical constant pi ||

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 || || `typeof(object)` || string || The type of `object`, `int`, `string`, `file` etc || || `size(object)` || int || Returns the objects 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 `size()` method or `size` or `length` attribute of the object in question. || || `sizeof(object)` || int || same as `size(object)`, see above || || `null(object)` || bool || Evaluates to `true` if object is the `nil` entity; otherwise `false`. This function can be used to check whether a reference (var) or property exists. Operator `empty` cant be used for this task, cause `empty` returns `true` if either not existing or if literatly *empty* (for example the empty string). || || `file(object)` || 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. || || concat(object..) || string || Creates a string by concatenating all stringized objects. If no object is provided, the empty string is returned. || || list(object..) || list || Returns a list where the lists elements consists of the objects provided. If no objects are provided, the empty list is returned. || || append(object..) || list || This function is similar to `list`. 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 `append('a,list('b'),'c')` evaluates to list `('a','b','c')` ||

Some mathematical functions are defined as well:

|| sin(double) || double || The mathematical sine function || || cos(double)

|| 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 larget 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. ||

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.

Primitve Types

Primite 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 specfies an not existing element, `null` is returned.

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 `.` and `..` 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.isdir
```

In this example of an EL expression, string `myfolder` is converted in a File object using property `tofile`. In addition, the so created object is checked whether it is a folder or not.

The following *properties* are defined on File objects: || Property || Type || Description || || `parent` || File || parent of file or folder as file object || || `toabs`|| File || file or folder as absolute file object || || `exists` || bool || check whether file or folder exists || || `isfile` || bool || check whether a file || || `isdir` || bool || check whether a folder (directory) || || `ishidden` || bool || check whether a hidden file or folder || || `isread` || bool || check whether a file or folder is readable || || `iswrite` || bool || check whether a file or folder is writable || || `size` || int || number of bytes in a (existing) file; 0 otherwise|| ||

`length` || int || same as `size` || || `mtime` || Date || last modification date || || `list` || File[] || array of files in folder ; otherwise `null` || || `tostr` || String || file name as string object || || `touri` || URI || file as URI object || || `tourl` || URL || file as URL object || || `delete` || bool || deletes the file or folder (true); false otherwise || || `mkdir` || bool || creates the folder (and intermediate) folders (true); false otherwise ||

Matcher Properties

A *matcher object* is created by task [Tasks#switch 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 || || `start` || int || The position within the input where the match starts. || || `s` || int || Same as `start` || || `end` || int || The position within the input where the match ends (the character at `end` is the last matching character) || || `e` || int || Same as `end` || || `groups` || int || The number of capturing groups in the (regular) expression. || || `size` || int || Same as `groups` || || `length` || int || Same as `groups` || || `n` || int || Same as `groups` || || `pattern` || string || The regular expression that was used for this match. Notice that glob expressions are translated into regular expressions. || || `p` || string || Same as `pattern` || || *i* || matcher || The matcher object for *i*'th capturing group. See task [Tasks#switch switch] for examples. ||

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` || || file || if the file exists || `false` otherwise || || string || if string is empty || `false` otherwise || || null || never || always || || boolean || if true || otherwise || || *other* || always || never ||

Part II

here Im listing all task, types and macros.

let

Then meaning of `null` and *void*

|| 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` ||

To illustrate, 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>
```

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 || [EL] || 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">
 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
 hello, #{ 'world' }
</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 `el="false"` does not prohibit you from using [EL#EL_References EL references].

Further Links

- Javadoc
- Source

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.