Syntax specification

Moodle High Level Script Engine

Syntax basics

The MoodleScript syntax aims to be simple, so we will avoid excess of formal syntactical forms and will prefer a keyword based language that results in naturally readable statements.

A moodle command is basically composed by keyword, arguments, variables and parameter lists. All terms must be separated by at least one spacing char. A command always starts with a verb (ADD, REMOVE, ENROL, BACKUP, etc.) and follows with an alternance of keywords and arguments, forming a readable sentence. Some commands may accept an argument list, f.e. to provide a set of attributes to an object to create, or giving conditional clauses for an object destruction.

The common form of a MoodleScript command is:

```
VERB KEYWORD1{1,n} [ [arg1]{0,n} KEYWORD1{0,n} ]{0,n}
```

A command that accepts an argument list as last parameters is of the form:

```
VERB KEYWORD1{1,n} [ [arg1]{0,n} KEYWORD1{0,n} ]{0,n} HAVING
key1: value1
[key2: value2]
...
[keyn: valuen]
```

Only one single argument list can be given for a single command.

Keywords

Keywords are reserved words that have a special syntactic meaning when present at some location in the command. Keywords are ALWAYS UPPERCASE.

Keywords are:

Verbs (always first command word)

Verbs are single tokens (one word) and must designate an action (e.g ADD or REMOVE)

Accepted verbs:

```
ADD, REMOVE, ENROL, BACKUP, MOVE
```

Object types

Object typenames designate a class of objects to administrate. They can be single words or multiple words (in a set of known expression).

```
COURSE, CATEGORY, ENROL METHOD, USER, COHORT, BLOCK, MODULE, etc.
```

Context articulations

context articulations are small words that tell what is the use of an argument, in order to provide naturzal readability of the script command.

```
IN, FOR, TO, IF EXISTS, IF NOT EXISTS
```

Arguments, identifiers and variables

Arguments are usually moodle object identifiers, terminal values or eventually global variables, for finding or setting values. the nature of the argument will vary across the syntax, and refers usually to the most common or trivial object type that is expected in the syntax.

F.E., for en enrolment syntax:

```
ENROL id:33 IN id:3 AS shortname:student USING manual
```

shows 4 attributes that are naturally refering to (successively) a user, a course, a role and an enrol method.

In case the expression have some possible ambiguity, additional keywords will be used to discriminate possible cases.

Identifiers

when the syntax requires to identify an existing object, and this object may be identified by several information, we will use an explicit field discriminator and value couple, f.e. for a user, there are usually 4 admitted possible identifiers as primary id, username, idnumber or email.

Thus the following identifiers are usable when searching for a user :

```
id:33
username:johndoe
idnumber:JD@35465
email:john.doe@gmail.com
```

Special identifier case : identifier given by a function

In some applications, we want an identifier being given by a custom or existing function, depending on some current context. the identifier form will accept the 'func' prefix to identifiy some plugin function to call to get an identifier:

Example:

```
 idnumber:func:local_ent_installer@get_teacher_cat_idnumber
```

will invoke the function `local_ent_installer_get_teacher_cat_idnumber()` in the plugin `local_ent_installer' locallib.php (or by default, lib.php'')` local library to get the expected identifier. the result of the function will be used as idnumber to find ethe primary identifier of the objet.

You cannot pass any parameters to this call, so the identifier must be fully determined using current environment globals such as $USER, $COURSE, etc. to compute the expected identifier.

Here is a sample of an application function that computes the current user's owned category to move a course in:

```
  /**
   * Provides an uniform scheme for a teacher category identifier.
   * @param object $user a user object. If user is not given will return the
cat identifier of
   * the current user.
   * @return string
   */
  function local_ent_installer_get_teacher_cat_idnumber($user = null) {
      global $USER;

      if (is_null($user)) {
          $user = $USER;
      }

      $teachercatidnum =
strtoupper($user->lastname).'_'.substr(strtoupper($user->firstname), 0, 1).
      $teachercatidnum .= '$'.$user->idnumber.'$CAT';
      return $teachercatidnum;
   }
```

Called in a moodlescript stack context, it will compute the category idnumber of the current user, so we can write a moodlescript move instruction as follows, moving the current course to the adequate destination:

```
  MOVE COURSE current TO
idnumber:func:local_ent_installer@get_teacher_cat_idnumber
```

Literal Argument

Literal arguments are simple words or strings. There is at the moment a restriction on syntax as strings are not specifically delimited (or only delimited by keywords). So strings should not contain keywords expressions. this is likely why we chose keywords in strict uppercase, to minimize syntactic collision with literal usual strings.

Variables

We may need to inject some environmental values in the script to replace some non terminal placeholders. An execution stack can be fed at launch time with a global context data stub that will be merged with each instruction local context (adding or overriding values). Global context variables can be placed whereever in script statements or attribute lists using the Moodle common SQL named variable form:

```
:varname
```

To be valid, the placeholder expression MUST have at least one space character before it.

You may obtain a list of the available variables in the stack logger using the following instruction:

```
LIST GLOBALS
```

this will output, e.g. in the admin tool console, giving the console environment preset variables:

```
> GLOBAL CONTEXT
> wwwroot: http://dev.moodle31.fr
> currentuserid: 2
> currentusername: admin
> siteshortname: DEV31
```

Any local invocation of a MoodleScript stack may run the stack with his own global environment variable set, to serve some specific component scoped scripting needs.

Special keywords (metas)

'**current**'

'current' is a special keyword in place of an expected identifier that will resolve into the nearest current object in the executing environment. F.e, if the expected object is a user identifier, current will resolve to $USER->id. If 'current' addresses a course identifier, it wil usually resolve as $COURSE→id, unless another course id is given to the execution stack by the global context (for plugin developers).

The use of current will simplify scripts run within a known context, by using shorten expressions:

```
ADD ENROL METHOD guest TO current
```

For adding an enrolment method to the current course.

```
ENROL current INTO current AS student
```

For enrolling the current user (the $USER being executing the script) into the current course.

'**last**' or '**first**'

this usually addresses the first available or last available item in the current syntax context. this is used f.e. for blocks location in a region, but might also address any object locaton being sorted with a sortorder attribute.

'**runtime**'

Usually identifiers and variable can be evaluated at parse time or at check time, because they are litterals in the script, or they come from some input or global context. But this is not true in all cases. Lets take an example:

In the following scriptlet:

```
ADD CATEGORY "New category" TO idnumber:EXISTINGCAT HAVING
idnumber: NEWCAT

MOVE COURSE idnumber:SOMECOURSE TO idnumber:NEWCAT
```

We run into an issue because at parse time or at check time, NEWCAT category is not yet created. Thus we must tell the engine that in the second statement, we need the engine waiting the latest moment to evaulate the identifier to move the course in.

This can be done by the special keywork runtime: and we'll rewrite the scriplet as follows:

```
ADD CATEGORY "New category" TO idnumber:EXISTINGCAT HAVING
idnumber: NEWCAT

MOVE COURSE idnumber:SOMECOURSE TO runtime:idnumber:NEWCAT
```

Using the runtime: special keyword will prevent the parser to evaluate and resolve the identifier, and will store it's initial definition in the handler class. The handler will also NOT try to resolve it at check time, as check time only checks the conditions of execution of all the statements without executing them. At real execution time, the identifier will be resolved to get it's definitive actual value.

Note that 'runtime' variables may raise a real error situation that cannot be recovered or anticipated by the engine and may terminate in a tecnhical failure of the script and the calling process.

Return to component index

From:
https://docsen.activeprolearn.com/ - **Documentation Moodle ActiveProLearn**

Permanent link:
https://docsen.activeprolearn.com/doku.php?id=local_retained:moodlescript:syntaxspecification

Last update: **2026/01/13 07:58**