JOD z Interface Words

---

compj	compress J code	gt	get edit window text
del	delete objects	make	generates dictionary scripts
did	dictionary identification	newd	create a new dictionary
disp	display dictionary objects	od	open dictionaries
dnl	dictionary name lists	packd	backup and pack dictionaries
doc	format word comments	put	store objects in dictionary
dpset	set and change parameters	regd	register dictionaries
ed	edit dictionary objects	restd	restore backup dictionaries
et	put text into edit window	revo	list recently revised objects
get	get objects	rm	run macros
globs	global references	rtt	run tautology tests
grp	create and modify groups	uses	return word uses

prev **toc** **next**

Master File - `jmaster.ijf`

Master file documentation - NIMP

prev **toc** **next**

---

codes – JOD argument codes

The left, and some right, arguments of dictionary verbs are specified with object, qualifier and option codes. Object codes are typically the first argument code while options and qualifiers usually occupy the second and third positions. Options and qualifiers are sometimes negative. Negative values modify the codes. The examples following the code tables will make this clear.

Object Codes

Noun	Value	Use	Example
WORD	0	word code	`0 dnl ''  NB. list all words on path`
TEST	1	test case code	`1 put 'test'  NB. put test`
GROUP	2	group code	`2 put 'group header ...'  NB. put group header`
SUITE	3	suite code	`3 get 'suite'  NB. get suite members, list of test names`
MACRO	4	macro code	`4 disp 'test'  NB. display macro`

Qualifier Codes

Noun	Value	Use
DEFAULT	7	default action
EXPLAIN	8	short explanation text code
DOCUMENT	9	documentation text code
NVTABLE	10	name value table code
REFERENCE	11	reference code
JSCRIPT	21	J script code
LATEX	22	LaTeX text code
HTML	23	HTML text code
XML	24	XML text code
UTF8	25	Unicode UTF8 text code

Option Codes
Value Use

1, -1 option one (-) is a modifier
2, -2 option two
3, -3 option three

Context dependent. The meaning depends on the verb for example for dnl, (dictionary name list), the option code specifies prefix, suffix and contains pattern name searches. Negative codes specify the same search pattern but request that the result be returned as a path order list.

Code examples:

```
3 get 'test'                NB. get suite members, list of test names
1 put 'testname';tvalue     NB. char list tvaluestored as test with
testname
2 del 'group'               NB. group with name group deleted
```

prev **toc** **next**

---

compj – compress J code

`compj` compresses J code by removing comments, white space and shortening safe local identifiers to single characters. Code compression is useful when preparing production scripts. The JOD system script

```
~addons\general\jod\jodnws.ijs
```

is an example of a compressed J script. In it's fully commented form this script is about 168 kilobytes when squeezed with `compj` it shrinks to about 65 kilobytes. `compj` does not compress words in JOD dictionaries it returns a compressed script result.

WARNING: to effectively use `compj` you must understand how to mark ambiguous names. If you do not correctly mark ambiguous names `compj` compression will break your code!

Prior to compressing a word apply `globs` to expose any name problems.

Ambiguous names in J are words created in object instances, temporary locale globals, names masked by indirect assignments and objects created with execute. When you use ambiguous names augment your code with sufficient information to clearly resolve and cross reference all names. JOD provides two comment scope tags `(*)=.` and

`(*)=:` to clarify ambiguous names.

1. *NB. (*)=. local names declared after tag*
2. *NB. (*)=: global names also declared*

The following examples illustrates how to use these tags:

```
indirectassignments=: 4 : 0


NB. Indirect assignments ()=: create objects that
NB. elude static cross referencing. Declaring
NB. the names global and local makes it possible
NB. to cross reference this verb with globs
globs=.   ;:'one two three'
(globs)=: y     NB. names after tag declared global (*)=: one
two three
locs=.    ;:'we are hidden locals'
(locs)=.  i. 4  NB. (*)=. we are hidden locals


NB. names seemingly used out of nowhere - EVIL AND CRUEL!
one * two * three
we + are + hidden + locals
)
```

With great power comes great responsibility!

```
createobject=: 3 : 0


NB. Object initialization often creates global nouns that
NB. are not really globals. They only exist within the
NB. the scope of the object. Tags can over ride J's
NB. global scope for cross referencing purposes.
NB. create all sorts of "global" stuff in an object
THIS=: STUFF=: IS=: INSIDE=: AN=: OBJECT=: 1


NB. over ride J's scope by declaring all these names
NB. local despite the global assignments in the code. I
```

```
NB. put ! just before the local tag to indicate this trick
NB. !(*)=. THIS STUFF IS INSIDE AN OBJECT
1
)
```

More examples of the use of comment scope tags can be found in the fully commented JOD source code. JOD source code is not distributed with JOD. It can be downloaded from The JOD Pages. JOD source is distributed as JOD dictionary dump scripts.

Monad: *compj clName | blclNames*

```
compj 'squeezeme'  NB. compress a single dictionary word
compj }. dnl 'fat' NB. compress all words beginning with 'fat'
```

```
NB. Compress all words in a group. The result is a JOD standard
(rc;value)
NB. where the value is a character list compressed J script.
'rc script'=. compj }. grp 'group'
```

prev **toc** **next**

---

del - delete objects

`del` deletes dictionary objects. If objects are on the search path but not in the put dictionary nothing will be deleted and the *non-put-dictionary* objects will be identified in an error message.

Warning: `del` will remove objects that are in use without warning. This can lead to broken aggregates. For example: if a word, that belongs to a group, is deleted the group is broken. An attempt to `get` or `make` a broken group or suite will result in an error.

Monad: *del clName | blclNames*

```
del 'word'                        NB. delete one word
del 'go';'ahead';'delete';'us' NB. delete many words
```

Dyad: *iaObject del clName | blclNames*

```
1 del 'test'  NB. toast a test
2 del 'group' NB. group deleted—all words remain in dictionary

       NB. but the “grouping” disappears


2 del ;:'we are toast'  NB. delete many groups
3 del 'suite'           NB. delete suites and macros
4 del 'macro'
4 del 'macro';'byebye'  NB. delete many macros
11 del ;:'remove our references'  NB. delete references
```

prev **toc** **next**

---

did - dictionary identification

`did` identifies the current open dictionaries.

Monad: *did uuIgnore*

```
did 0  NB. identify current dictionaries - lists open
dictionaries in path order
```

Dyad: *uuIgnore did uuIgnore*

```
0 did 0   NB. identify current open dictionaries and show basic
statistics
```

```
did~ 0    NB. handy idiom
```

prev **toc** **next**

---

disp – display dictionary objects

`disp` displays dictionary objects. `disp` returns a character list when successful and the standard boxed (rc;message) when reporting errors.

Monad: *disp clName|blclNames*

```
disp 'word'      NB. display a single word
```

```
disp ;:'go ahead show us'  NB. display many words
```

Dyad: *iaObject disp clName|blclNames*

(iaObject,iaOption) disp clName|blclNames

```
1 disp 'test'  NB. show a test
```

```
2 disp 'group' NB. generate and display a group

2 1 disp 'groupheader'      NB. display the group text or header

3 disp 'suite' NB. generate and display a suite
```

```
3 1 disp 'suiteheader'     NB. display the group text or header


4 disp 'macro'             NB. display one macro
4 disp 'macro';'byebye'    NB. display many macros
```

prev toc next

---

dnl - dictionary name lists

`dnl` searches and returns dictionary name lists. The entire path is searched for names and duplicates are removed. A negative option code requests a path order list. A path order list returns the objects in each directory in path order. Raising, removing duplicates and sorting a path order list gives a standard `dnl` list. `dnl`'s arguments follow the pattern:

```
(n,p,[d]) dnl 'str' NB. n is one of 0 1 2 3 4

NB. p is one of 1 2 3 _1 _2 _3

NB. optional d is word name class or macro type
```

Monad: *dnl zl | clPstr*

```
dnl '' NB. list all words on current dictionary path
dnl 'prefix' NB. list all words that begin with prefix
```

Dyad: *iaObject dnl zl | clPstr*

(iaObject,iaOption) dnl zl | clPstr
(iaObject,iaOption,iaQualifier) dnl zl | clPstr

```
0 dnl ''  NB. all words (same as monad)
1 dnl ''  NB. list all tests
2 dnl ''  NB. list all groups
3 dnl ''  NB. list all suites
4 dnl ''  NB. list all macros
```

```
NB. A word can appear in two dictionaries. When getting such a
word the first

NB. path occurrence is the value returned. The second value is
shadowed by the first.

NB. As only one value can be retrieved dnl returns unique name
lists.
0 1 dnl 'str'  NB. match word names beginning with str
0 2 dnl 'str'  NB. match word names containing the string str
0 3 dnl 'str'  NB. match word names ending with string str


NB. words and macros have an optional third item that denotes
name class or type
0 1 1 dnl 'str'  NB. adverb names beginning with str
0 1 3 dnl 'str'  NB. verb names containing str
0 2 0 dnl 'str'  NB. nouns ending with str
4 1 21 dnl 'jscript' NB. J macro names beginning with jscript
4 2 22 dnl 'latex'   NB. LaTeX macro names containing latex
4 3 23 dnl 'html'    NB. HTML macro names ending with html



NB. A negative second item option code returns a path order
list
0 _1 1 dnl 'str' NB. nouns beginning with str (result is a list
of lists)
2 _2 dnl 'str'   NB. group names containing str
3 _3 dnl 'str'   NB. suite names ending with str
```

prev **toc** **next**

---

doc – format word comments

`doc` formats the leading comment block of explicit J words. The comment block must follow J project manager scriptdoc conventions. The comment style processed by `doc` is illustrated in the following example. More examples of `doc` formatting can be examined by displaying words in the distributed JOD dictionaries.

```
docexample0=: 3 : 0
NB.*docexample0 v-- the leading block of comments
NB. can be a scriptdoc compatible mess as far
```

```
NB. as formatting goes.
NB.
NB. However, if you run doc over
NB. a word in a JOD dictionary your mess is cleaned up. See
below.
NB. monad: docexample uuHungarian
NB.
NB. text below MONAD and DYAD marks is left intact
NB. this region is used to display example calls
J code from now on
)


docexample0=:3 : 0
NB.*docexample0 v-- the leading block of comments can be a
NB. scriptdoc compatible mess as far as formatting goes.
NB.
NB. However, if you run doc over a word in a JOD dictionary
your
NB. mess is cleaned up. See below.
NB.
NB. monad: docexample uuHungarian
NB.
NB. text below MONAD and DYAD marks is left intact
NB. this region is used to display example calls
j code from now on
)
```

Monad: *doc clName*

```
doc 'formatme' NB. format leading comment block
```

prev **toc** **next**

---

dpset - set and change parameters

`dpset` modifies dictionary parameters. JOD uses a variety of values that control putting, getting and generating objects. Dictionary parameters are stored in individual dictionaries and the main master file. Master file parameters are initially set from the `jarparms.ijs` file and cannot be reset without editing `jarparms.ijs` and recreating the master file. Individual dictionary parameters can be changed at any time. `dpset` is permissive. It will allow parameters to be set to any value. Invalid

values will crash JOD! Before setting any values examine the jarparms.ijs file. This file is used to set the default values of dictionary parameters.

Note: If you accidentally set an invalid parameter value you can recover using dpset's DEFAULTS option.

Not all dictionary parameters can be set by dpset. The parameters dpset can change are dictionary specific user parameters. There are a number of system wide parameters that are set in code and require script edits to change.

If JOD or the host OS crashes the master file could be left in a state that makes it impossible to reopen dictionaries. RESETME and RESETALL clears the read status codes in the master file. RESETME resets all dictionaries recently opened from the current machine. RESETALL resets all dictionaries in the master file. In the worst case you can rebuild the master file with the script resetjod.ijs.

Monad: *dpset zl | clName | (clName;uuParm)*

```
dpset ''             NB. list all parameters and current values
dpset 'DEFAULTS'     NB. restore default settings in put dictionary
```

```
NB. option names are case sensitive
dpset 'RESETME'      NB. resets current machine dictionaries.
dpset 'RESETALL'     NB. resets all dictionaries


NB. Note: if a JOD dictionary is being used by more than one
user never use
NB. RESETALL unless you are absolutely sure you will not reset
other users!


dpset 'CLEARPATH'    NB. clears the put dictionary reference
path
dpset 'READONLY'     NB. makes the current put dictionary read-
only
dpset 'READWRITE'    NB. makes the current put dictionary read-
write
dpset 'GETFACTOR';1000     NB. get 1000 objects in each get loop
pass
```

ed – edit dictionary objects

ed fetches or generates dictionary objects and puts them in an edit window for editing.

Monad: *ed clName | blclNames*

```
ed 'word'   NB. retrieve word and place in edit window
ed ;:'many words edited' NB. put all words in edit window
```

Dyad: *iaObject ed clName | blclNames*
(iaObject,iaOption) ed clPstr

```
1 ed 'test'    NB. edit test
2 ed 'group'   NB. generate group and place in edit window
3 ed 'suite'   NB. generate test suite and place in edit window
4 ed 'macro'   NB. edit macro text
2 1 ed 'group'  NB. edit group header text
3 1 ed 'suite'  NB. edit suite header text
```

et – put text into edit window

Monad: *et clText*

```
et 'put text in edit window'
```

```
et read 'c:\temp\text.txt'
```


get - get objects

`get` retrieves dictionary objects and information about dictionary objects. There is a close correspondence between the arguments of `get` and `put`. A basic JOD rule is that if you can `put` it you can `get` it and vice versa.

Monad: *get clName | blclNames*

```
get 'word'     NB. get word and define in current locale
get }. grp ''  NB. get a group
```

Dyad: *ilOptions get clName | blclNames*
clLocale get clName | blclNames

```
0 get 'word'  NB. get word (monad)
0 7 get ;:'words are us'  NB. get words (monad)
```

```
NB. for words a character left argument is a target locale
'locale' get ;:'hi ho into locale we go'  NB. get into locale
```

```
'666' get ;:'beast code'     NB. allow numbered locales
0 8 get ;:'explain us ehh'   NB. explain words
0 9 get ;:'document or die' NB. word documentation
0 10 get 'define';'not'      NB. get word scripts without
defining


NB. information about stored words can be retrieved with get
0 12 get ;:'our name class' NB. J name class of words
0 13 get ;:'our creation'    NB. word creation dates
0 14 get ;:'last change'     NB. last word put dates
0 15 get ;:'how big are we' NB. word size in bytes
1 7 get 'i';'test';'it'      NB. get test scripts
1 8 get ;:'explain tests'    NB. test explanations
1 9 get 'radical'            NB. test case documentation


NB. information about stored tests
1 13 get ;:'our creation'    NB. test creation dates
1 14 get ;:'last change'     NB. last test put dates
1 15 get ;:'how big are we' NB. test size in bytes
2 7 get ;:'groupies cool'    NB. get group scripts
2 8 get 'group';'explain'    NB. get group explain text
2 9 get 'document'           NB. get group document text
3 7 get ;:'this suites me'         NB. suite text
3 8 get ;:'suites need comments'  NB. explain suites
3 9 get ;:'document your suites'  NB. document suites
4 get 'jmacro';'html';'latex'      NB. get various macros
4 8 get ;:'macros need explaining'  NB. explain
4 9 get ;:'and documents too'        NB. document
```

prev **toc** **next**

---

globs - global references

`globs` analyzes global references in words and tests. A global reference is a nonlocal J name where nonlocality is with respect to the current word's scope. Names with locale references, for example:

1. `jread_jfiles_` direct locale reference
2. `did__jd2` indirect locale (object) reference
3. `boo__hoo__too` two levels of indirection

are treated like primitives. This makes it possible to define clean locale/object

interfaces. In the case of indirect locale references the suffix noun must exist to determine the name class of the word. This makes static name analysis difficult. By treating such references as "primitives" this problem is neatly swept under the proverbial rug.

For example the jfiles utility is typically accessed through z locale definitions like:

```
jread_z_ =: jread_jfiles_
```

Words that use `jread` can simply call it without any locale suffixes. For this case `globs` will detect the use of `jread` but will cease searching the call tree when it encounters `jread_jfiles_`

Globals referenced by test scripts are not stored because tests often manipulate their working environments in ways that make static name analysis unfeasible. `globs` is one of two verbs, (`globs`, `grp`), that create references. For `globs` to store references the word must be in the put dictionary, all word references must exist on the path and the current path must match the put dictionary path.

Monad: *globs clName*

```
globs 'word' NB. list globals in locale word
```

Dyad: *iaObject globs clName*

```
NB. update globals referenced in word uses stored word text
```

```
0 globs 'word'


0 globs&> }. grp 'group' NB. update all words in a group
1 globs 'test'           NB. list global references in test
text


NB. result may be misleading depending on how the test
manipulates its environment

NB. returns a boxed table classifying all name references in
locale word.
```

```
11 globs 'word'
```

prev toc next

---

grp - create and modify groups

grp creates and modifies word groups and test suites. A group is a list of objects. Operations on groups do not change the objects that belong to groups. When a group is created the put dictionary's reference path is compared to the current dictionary path. If the paths do not match an error is returned and the group is not created.

Monad: *grp zl | clName | blclNames*

```
grp ''       NB. list all word groups (2 dnl '')
grp 'group'  NB. list words in group
```

```
NB. create/reset group—first name is the group name
grp 'group';'list';'of';'group';'names'

NB. has effect of emptying but not deleting group



grp <'group'
```

Dyad: *iaObject grp zl | clName | blclNames*

```
3 grp ''      NB. list all test groups (suites) (3 dnl '')
3 grp 'suite' NB. list tests in suite
2 grp 'group';'list';'of';'group';'names'  NB. (monad)
3 grp 'suite';'list';'of';'test';'names'   NB. create/reset suite
3 grp <'suite' NB. empty suite
```

prev toc next

---

gt – get edit window text

Monad: *gt zl | clName*

```
gt 'word' NB. returns text from the word.ijs edit window
```

```
NB. using gt to update a test and macro.
1 put gt 'test';gt 'test'

4 put gt 'macro';21;gt 'macro'
```

prev **toc** **next**

---

make - generates dictionary scripts

make generates J scripts from objects stored in dictionaries. The generated scripts can be returned as results or written to file. Generated scripts are stored in the standard dump, script and suite subdirectories. Monadic make dumps all the objects on the current path to a J script file. The dump file is a single J script that can be used to rebuild dictionaries. make uses the reference path to generate words, tests, groups and suites. When generating aggregate objects make returns an error if the current path does not match the reference path. By default dyadic make generates objects that exist in the current put dictionary. This can be overridden with a negative option code.

Monad: *make zl | clDumpfile*

```
NB. Dump objects on current path to put dictionary dump directory.
NB. The name of the put dictionary is used as the dump file name.
make ''
```

```
make 'c:\dump\on\me.ijs' NB. dump to specified file
```

Dyad: *iaObject make zl | clName | blclNames*

(iaObject,iaOption) make clName

```
0 make ;:'an arbitrary list of words into a script'
0 2 make ;:'generate a character list script result'


2 make 'group' NB. make J script that defines a group
3 make 'suite' NB. make J script that defines a suite



NB. an option code controls whether results are written to file
(1 default)
NB. or returned (2) for word lists, groups and suites. Default
NB. dictionary file locations are the subdirectories created by
newd.
2 2 make 'group'  NB. make and return group script
3 1 make 'suite'  NB. make put dictionary suite script and
write to file


NB. make and file group script. The group does not

NB. have to exist in the put dictionary but can
NB. occur anywhere on the path.

2 _1 make 'group'
3 _1 make 'suite' NB. make suite script and write to file
```

prev **toc** **next**

---

newd - create a new dictionary

`newd` creates a new dictionary. Dictionary creation generates a set of files in a standard dictionary directory structure. The root directory, dictionary name, and optional dictionary documentation can be specified. All other dictionary creation parameters are taken from the master file.

Monad: *newd clDictionary*

newd (clDictionary;clPath)
newd (clDictionary;clPath;clDocumentation)

```
NB. if no location is specified the dictionary is created in the
default directory
newd 'makemydictionary'
newd 'new';'c:\location\' NB. create with name in location
```

```
NB. optional third item is dictionary documentation
newd 'new';'c:\location\';'Dictionary documentation ...'
```

prev **toc** **next**

---

od - open dictionaries

od opens dictionaries. Open dictionaries are appended to the path in the order they are opened. Dictionaries can be opened READWRITE (default) or READONLY. Only one J task can open a dictionary READWRITE. Any number of tasks can open a dictionary READONLY. If any task has a dictionary open READONLY it can only be opened READONLY by other tasks. If a dictionary is opened READWRITE by a task it cannot be opened by other dictionary tasks. This harsh protocol insures that only one task can update a dictionary.

The first dictionary on the search path is special! It is the only dictionary that can be updated by JOD verbs. Because most updates are put's the first dictionary is called the put dictionary.

Monad: *od zl | clDictionary | blclDictionaries*

```
od '' NB. list registered dictionaries
```

```
NB. names must match exactly
od 'dictionary'    NB. open read/write
od 'd1';'d2';'d3'  NB. opens di read/write
```

Dyad: *iaOption od zl | clDictionary | blclDictionaries*

```
```

```
1 od '' NB. list registered dictionaries (monad)
3 od '' NB. close all open dictionaries (related to did 4)
```

```
1 od 'dictionary'   NB. open read/write (monad)
2 od 'dictionary'   NB. open read only and append to any path



2 od 'd1';'d2';'d3' NB. open di read only and append to any
path
3 od ;:'d0 d1 d2'   NB. close dictionaries and remove from path
4 od ''             NB. list all dictionaries and locations of
root directories
```

prev **toc** **next**

---

packd – backup and pack dictionaries

`packd` removes all unused space from dictionary files by copying active components to new files. After the `packd` operation is complete the new dictionary files are renamed to match the original files. During the copy operation directories are checked against the items in dictionary files. If a *directory–data–discrepancy* is detected the pack operation ends with an error. Old files are renamed with an increasing sequential backup number prefix, e.g.: `13jwords.ijf` and retained in the backup subdirectory. If a `packd` operation succeeds the backup dictionary has no directory data inconsistencies.

A packd operation can be reversed with `restd`. ***There is no JOD facility for deleting backup files. To erase backup files use OS facilities.***

The read/write status of a dictionary is recorded in the master file. JOD assumes all users and tasks point to the same master file.

Monad: *packd clDictionary*

```
od 'dictionary'    NB. packd requires a read/write open.

packd 'dictionary' NB. reclaim unused file space in dictionary
                   NB. and retain original files as a backup
```

put - store objects in dictionary

The put verb stores objects in the put dictionary. It can store words, tests, groups, suites and macros. As a general rule: if something can be stored with put it can be retrieved by get.

Monad: *put clName | blclNames*

```
put 'word' NB. default is put words from base locale
```

Dyad: *iaObject put clName | blclNames | btNvalues*
clLocale put clName | blclNames | btNvalues
(iaObject,iaQualifier) put clName | blclNames | btNvalues

```
0 put ;:'w0 w1 w2 w3 w4'     NB. put words (monad)
'locale' put 'w0';'w2';'w3' NB. put words from specified locale
'99' put 'word'              NB. numbered locales
```

```
NB. put explain/document text—words must exist in dictionary
0 8 put (;:'w0 w1'),.('text ...';'text ...')
0 9 put (;:'w0 w1'),.('text ...';'text ...')


NB. put words from name class value table
0 10 put ('w0'; 'w1'),.(3;3),.'code0..';'code1..


NB. put tests from name value table
1 put (;:'t0 t1'),.('text ...';'text ...')


NB. put test explain/document text
1 8 put (;:'t0 t1'),.('text ...';'text ...')
1 9 put (;:'t0 t1'),.('text ...';'text ...')


NB. put group scripts from name,value table
NB. A group script is an arbitrary J script that preceeds the
code generated by make.
```

```
2 put (;:'g0 g1'),.('text ...';'text ...')


NB. put group explain/document text
2 8 put (;:'g0 g1'),.('text ...';'text ...')
2 9 put (;:'g0 g1'),.('text ...';'text ...')


NB. put suite scripts from name value table
3 put (;:'s0 s1'),.('text ...';'text ...')


NB. put suite explain/document text
3 8 put (;:'s0 s1'),.('text ...';'text ...')
3 9 put (;:'s0 s1'),.('text ...';'text ...')


NB. put macro scripts from name, type, value table
4 put (;:'m0 m1'),.(21;21),.('text ...';'...') NB. J
4 put (;:'m0 m1'),.(22;22),.('text ...';'...') NB. LaTeX
4 put (;:'m0 m1'),.(23;23),.('text ...';'...') NB. HTML

4 put (;:'m0 m1'),.(24;24),.('text ...';'...') NB. XML

4 put (;:'m0 m1'),.(25;25),.('text ...';'...') NB. plain text
UTF-8




NB. put macro explain/document text
4 8 put (;:'m0 m1'),.('text ...';'text ...')
4 9 put (;:'m0 m1'),.('text ...';'text ...')
```

prev **toc** **next**

---

regd - register dictionaries

`regd` registers and unregisters dictionaries in the master file . A dictionary is a set of files in a standard directory structure. The `newd` verb creates JOD directories and files. *There is no client verb that destroys dictionaries; actual deletion of dictionary files and directories must be done using other means.* However, you can unregister a dictionary. When a dictionary is unregistered it is removed from the main dictionary directory in the master file. It will no longer appear on `od` lists and will no longer be accessible with JOD interface verbs. Conversely, you can also register dictionaries with regd.

Monad: *regd (clDictionary;clPath;clDocumentation)*

```
NB. register dictionary with name in directory and dictionary must
exist
regd 'name';'c:\location\'
```

```
NB. register dictionary with optional documentation
regd 'name';'c:\location\';'Documentation ...
```

Dyad: *iaOption regd clDictionary*

```
3 regd 'name' NB. unregistering a dictionary does not delete data files
```

```
NB. regd can be used to rename dictionaries and update
dictionary documentation
'name path' =. _2 {. 3 regd 'badname'  NB. unregister
doc =. 'brand spanking new documenation'



NB. re-register with new name and documentation

regd 'goodname';path;doc
```

prev **toc** **next**

---

restd – restore backup dictionaries

`restd` restores the last backup created by packd.

Monad: *restd clDictionary*

```
NB. open dictionary read/write - must be first dictionary on the path
od 'lastbackup' [ 3 od ''
```

```
NB. restore last dictionary backup
restd 'lastbackup'
```

prev toc next

---

revo – list recently revised objects

`revo` lists recently recently revised objects. Only `put` dictionary objects can be revised and only put operations are considered revisions.

Monad: *revo zl | clName*

```
revo ''     NB. all put dictionary words in last put order
revo 'boo'  NB. revised words with names beginning with 'boo'
```

Dyad: *iaObject revo zl | clName*

```
1 revo ''     NB. list all revised tests
3 revo 'boo'  NB. revised suites with names prefixed by'boo'
```

prev toc next

---

rm – run macros

A JOD macro is an arbitrary J script. `rm` fetches J macro scripts and runs them.

`rm` sets the current locale to `base` and starts executing macro scripts in `base`.

Monad: *rm cl | blclNames*

```
rm 'macro' NB. run my J macro
```

```
NB. run macros with names starting with 'DoUs'
rm }. dnl 'DoUs'
```

Dyad: *iaOption rm zl | clName | blclNames*

```
1 rm 'quiet' NB. run J script and suppress output
```

```
1 rm ;:'run silent run deep' NB. note the repeat
```

prev **toc** **next**

---

rtt – run tautology tests

`rtt` runs tautology test scripts stored in JOD dictionaries.

J has a built in test facility see: `(0!:2)` and `(0!:3)`. These foreigns run scripts and stop if the result deviates from arrays of 1's. This facility is used by J's developers and `rtt` applies it to dictionary test scripts.

`rtt` starts scripts in the base locale.

Monad: *rtt clName | blclNames*

```
rtt 'tautologytest'       NB. run test script as a tautology
rtt }. 3 grp 'testsuite'  NB. run all tautology tests in a suite
```

Dyad: *iaOption rtt clName | blclNames*

```
0 rtt 'tautologytest'     NB. same as monad
1 rtt 'silenttautology'   NB. run tautology test and suppress output
2 rtt 'plaintest'         NB. run plain test and display output
```

uses - return word uses

`uses` lists words used by other words. The lists are derived from the cross references generated by `globs`. The typical result of `uses` is a boxed table. Column 0 is a list of names and column 1 is list of pairs of boxed lists. Each boxed list pair contains nonlocale and locale global references.

When computing the uses union, (option 31), only nonlocale references are searched for further references. In general it is not possible to search locale references as they typically refer to objects created at runtime. In this system such references are treated as black boxes. It's important to know an object is being referenced even if you cannot peer inside the object.

Monad: *uses blclName | clName*

```
uses ;:'word globals' NB. list all words used by words(0 globs)
```

Dyad: *iaObject uses blclName | clname*

```
0 uses 'word'  NB. same as monad
31 uses ;:'all known words we call'   NB. uses union of word
```