JOD

J Object Dictionary

A CODE, TEST AND DOCUMENTATION DATABASE SYSTEM FOR J

Author:
John D. Baker
`bakerjd99@gmail.com`

Release:
0.3.7
May 30, 2008

Recent Version History			
Date	**Version**	**Author**	**Description**
May 30, 2008	0.3.7	John D. Baker	recent draft
March 30, 2008	0.3.5	John D. Baker	first draft

Table 1: Version History

Contents

1 What is JOD? **3**

1.1 JOD Classes 4

2 JOD Words **5**

2.1 `addgrp` – add words/tests to group/suite 5
2.2 `compj` — compress J code 5
2.3 `del` — delete objects 7
2.4 `delgrp` – remove words/tests from group/suite 8
2.5 `did` — dictionary identification 8
2.6 `disp` – display dictionary objects 9
2.7 `dnl` — dictionary name lists 10
2.8 `doc` – format word comments 11
2.9 `dpset` — set and change parameters 12
2.10 `ed` – edit dictionary objects 14
2.11 `et` – put text into edit window 15
2.12 `get` — get objects 15
2.13 `getrx` – get required to execute 17
2.14 `globs` — global references 18
2.15 `grp` — create and modify groups 19
2.16 `gt` – get edit window text 20
2.17 `hlpnl` – display short object descriptions 20
2.18 `jodage` — age of JOD objects 22
2.19 `jodhelp` — return help 22
2.20 `lg` – make and load group 23
2.21 `locgrp` – list groups/suites with word/test 23
2.22 `make` — generates dictionary scripts 24
2.23 `mls` — make load script 25
2.24 `newd` — create a new dictionary 26
2.25 `od` — open dictionaries 26
2.26 `packd` – backup and pack dictionaries 28
2.27 `put` — store objects in dictionary 28
2.28 `regd` — register dictionaries 30
2.29 `restd` – restore backup dictionaries 31
2.30 `revo` – list recently revised objects 31
2.31 `rm` – run macros 32
2.32 `rtt` – run tautology tests 33
2.33 `uses` — return word uses 33

3 JOD Directory and File Layouts **35**
3.1 Master File — `jmaster.ijf` 35
3.2 Words File — `jwords.ijf` 35
3.3 Tests File — `jtests.ijf` 35
3.4 Groups File — `jgroups.ijf` 37
3.5 Suites File — `jsuites.ijf` 37
3.6 Macros File — `jmacros.ijf` 40
3.7 Uses File — `juses.ijf` 40

4 JOD Distribution **40**

1 What is JOD?

JOD is a code, test and documentation database system for the *J programming language*.

J is a modern array oriented functional programming language. Programming in J has a charming and distinctive flavor. Tasks decompose into scores of tiny programs that are collectively known as *words*. JOD stores and organizes J words and other objects in a dictionary database: hence the name **J O**bject **D**ictionary.

Code dictionary databases are not new. Similar systems have been developed for many programming environments. Storing code in a database might strike you as obtuse. Why compromise the ease, portability, and broad support of standard source code text files? Believe me, there are good reasons.

- J encourages brevity. Microscopic programs, (words), accumulate rapidly. Short J words are often general purpose words. They can be used in many contexts. How are scores of terse words best employed? Copying and pasting words quickly becomes an error prone chore. Putting them in multiple scripts leads to lots of file searching and inclusions. Storing words in one big script can clutter your J session with many unused definitions.[1] The best way to reuse short definitions is to put them in a system like JOD and fetch as required.
- With JOD there is only *one definition* for a given word. When word copies are scattered throughout many files it's not always easy to find the current version.
- There are no significant limits on vocabulary size. Scripts can hold thousands of words but it's a nuisance to edit such large files.
- The *complete definition* of a word, (all code, use examples and test scripts), can be quickly examined. Good English dictionaries contain far more than definitions. There are etymologies, synonyms, usage comments and illustrations. Similarly, *literate* software documentation contains far more than source code. You should find descriptions of basic algorithms, remarks about coding techniques, references to published material, program test suites, detailed error logs and germane diagrams. Storing all this in source code comments would horribly clutter programs. A dictionary is where this material belongs.
- *Relationships between words* can be stored. Accurate word references make it easier to quickly understand code. This is especially true if references and documentation are linked.
- JOD facilitates the *generation of scripts and the distribution* of code. When I program with JOD I rarely write entire "load" scripts. I use JOD to generate and distribute J scripts. JOD can fetch and execute arbitrary J scripts so you can manage very elaborate generation and distribution procedures.

[1] I am not a fan of *rampant over-inclusion*. Over-inclusion occurs when you load an entire class and only use a tiny portion of it. Unused code is not harmless. It always confuses programmers.

1.1 JOD Classes

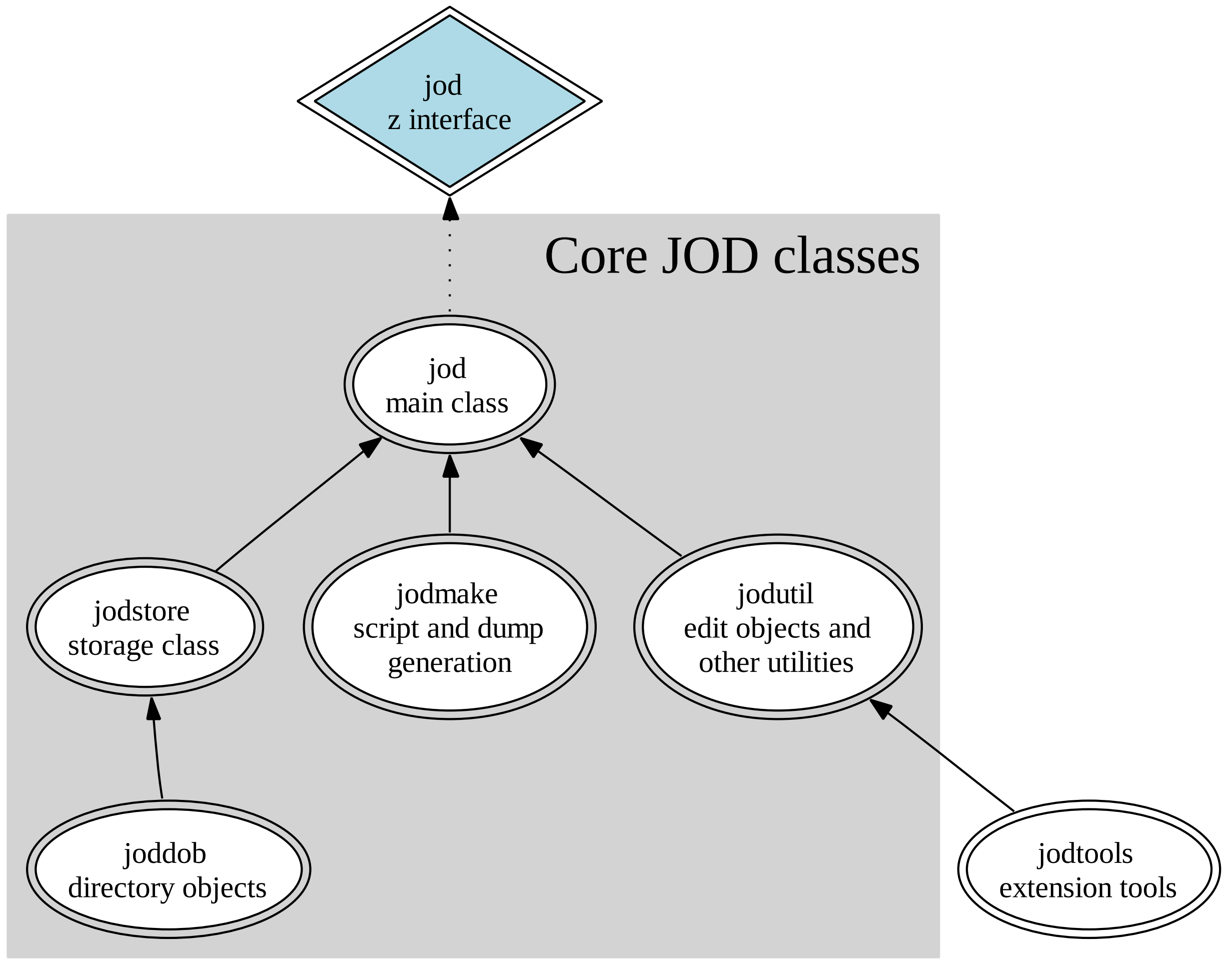

Figure 1: This diagram shows the relationship between JOD classes. JOD classes are represented with J locales or namespaces. The arrows indicate how names are resolved.

2 JOD Words

2.1 addgrp – add words/tests to group/suite

`addgrp` adds words to a group and tests to a suite.

```
Dyad:  clGroup addgrp clName ∨ blclNames
       (clSuite;iaObject) addgrp clName ∨ blclNames
```

```
NB. add a word to a group
'group' addgrp 'word'

NB. add many words to a group
'groupname' addgrp ;:'word names to group'

NB. boxed (x) is used for suites - 3 denotes suite
('suitename';3) addgrp ;:'tests added to suite'
```

2.2 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\jod.ijs
```

is an example of a compressed J script. In its fully commented form this script is about 168 kilobytes when squeezed with `compj` it shrinks to about 66 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`, see subsection 2.14, on page 18 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. local tag `NB. (*)=.  local names declared after tag`
2. global tag `NB. (*)=:  global names also declared`

The following examples illustrates how to use these tags:

```
indirectassignments=: 4 : 0

NB. Indirect assignments ()=: create objects
NB. that elude static cross referencing.
NB. Declaring the names global and local
NB. makes it possible to cross reference
NB. this verb with (globs)
globref=.  ;:'one two three'

NB. declared global (*)=: one two three
(globref)=: y

NB. declared local (*)=. we are hidden locals
locref=.   ;:'we are hidden locals'
(locref)=. i. 4

NB. without tags these names appear to
NB. used out of nowhere
one * two * three
we + are + hidden + locals
)
```

With great power comes great responsibility!

```
createobject=: 3 : 0

NB. Object initialization often creates
NB. global nouns that are not really globals.
NB. They only exist within the the scope of
NB. the object. Tags can over ride J's
NB. global scope for cross referencing.

NB. create  "globals" in an object
THIS=: STUFF=: IS=: INSIDE=: AN=: OBJECT=: 1

NB. over ride J's scope by declaring names local.
NB. !(*)=. THIS STUFF IS INSIDE AN OBJECT
1
)
```

More examples of the use of comment scope tags can be found in commented JOD source code. JOD source code is not distributed with JOD. You can get JOD source code by installing the jodsource addon or by downloading jodsource.zip from *The JOD Pages*. JOD source is distributed as JOD Dictionary Dump Scripts.

Monad: *compj clName ∨ blclNames*

```
NB. compress a single word
compj 'squeezeme'

NB. compress words beginning with 'fat'
compj }. dnl 'fat'

NB. Compress all words in a group.
'rc script'=. compj }. grp 'group'
```

2.3 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 groups and suites. Deleting a word that belongs to a group breaks the group: similarly for suites. An attempt to get or make a broken group or suite will result in an error. You can recover from this error by deleting references, (see below), and regrouping.

Monad: *del clName ∨ blclNames*

```
NB. delete one word
del 'word'

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

Dyad: *iaObject del clName ∨ blclNames*

```
NB. delete a test
1 del 'test'

NB. delete a group - words in the
NB. group are not not deleted
2 del 'group'
```

```
NB. delete many groups
2 del ;:'we are toast'

NB. delete suites and macros
3 del 'suite'
4 del 'macro'

NB. delete many macros
4 del 'macro';'byebye'

NB. delete references
11 del ;:'remove our references'
```

2.4 `delgrp` – remove words/tests from group/suite

`delgrp` removes words from a group and tests from a suite.

Removing objects from groups and suites does delete them. To delete objects use `del`.

Dyad: *clGroup delgrp clName ∨ blclNames*
(clSuite;iaObject) delgrp clName ∨ blclNames

```
NB. remove a word from a group
'group' delgrp 'word'

NB. remove many words from a group
'groupname' delgrp ;:'word names to remove'

NB. boxed (x) is used for suites - 3 denotes suite
('suitename';3) delgrp ;:'tests removed from suite'
```

2.5 `did` — dictionary identification

`did` identifies the open dictionaries.

Monad: *did uuIgnore*

```
NB. lists open dictionaries in path order
did 0
```

Dyad: *uuIgnore did uuIgnore*

```
NB. open dictionaries and basic statistics
0 did 0

NB. handy idiom
did~ 0
```

2.6 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*

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

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

Dyad: *iaObject disp clName ∨ blclNames*
(iaObject,iaOption) disp clName ∨ blclNames

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

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

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

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

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

NB. display one macro
4 disp 'macro'

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

2.7 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 arguments follow the pattern:

$(n, \langle p, \langle d \rangle \rangle)$ dnl 'str'

where:

n is one of `0 1 2 3 4`

optional p is one of `1 2 3 _1 _2 _3`

optional d is word name class or macro type

Monad: *dnl zl* $\vee$ *clPstr*

```
NB. list all words on current dictionary path
dnl ''

NB. list all words that begin with prefix
dnl 'prefix'
```

Dyad: *iaObject dnl zl* $\vee$ *clPstr*
(iaObject,iaOption) dnl zl $\vee$ *clPstr*
(iaObject,iaOption,iaQualifier) dnl zl
(iaObject,iaOption,iaQualifier) dnl clPstr

```
0 dnl ''  NB. all words (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
```

A word can appear in two dictionaries. When getting such a word the first path occurrence is the value returned. The second value is shadowed by the first. as only one value can be retrieved.

```
NB. match word names beginning with str
0 1 dnl 'str'

NB. match word names containing the string str
0 2 dnl 'str'
```

```
NB. match word names ending with string str
0 3 dnl 'str'

NB. words and macros have an optional third
NB. item that denotes name class or type

NB. adverb names beginning with str
0 1 1 dnl 'str'

NB. verb names containing str
0 1 3 dnl 'str'

NB. nouns ending with str
0 2 0 dnl 'str'

NB. J macro names beginning with jscript
4 1 21 dnl 'jscript'

NB. LaTeX macro names containing latex
4 2 22 dnl 'latex'

NB. HTML macro names ending with html
4 3 23 dnl 'html'

NB. A negative second item option
NB. code returns a path order list

NB. nouns beginning with str (result is a list of lists)
0 _1 1 dnl 'str'

NB. group names containing str
2 _2 dnl 'str'

NB. suite names ending with str
3 _3 dnl 'str'
```

2.8 doc – format word comments

`doc` formats the leading comment block of explicit J words. The comment block must follow J scriptdoc compatible 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. Incomplete dyad case not documented.

```
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
NB.     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
NB. your 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*

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

2.9 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 jodparms.ijs file and cannot be reset without editing `jodparms.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 `jodparms.ijs` file. This file is used to set the default values of dictionary parameters.

Note: If you 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 by:

1. Exiting J.
2. Deleting the files:

   ```
   ~addons\general\jod\jmaster.ijf
   ~addons\general\jod\jod.ijn
   ```

3. Restarting J.
4. Reloading JOD with: `load 'general/jod'`

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

```
NB. list all parameters and current values
dpset ''

NB. restore default settings in put dictionary
dpset 'DEFAULTS'

NB. option names are case sensitive

NB. resets current machine dictionaries
dpset 'RESETME'

NB. resets all dictionaries
dpset 'RESETALL'
```

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

```
NB. clears the put dictionary reference path
dpset 'CLEARPATH'

NB. makes the current put dictionary read-only
dpset 'READONLY'

NB. makes the current put dictionary read-write
dpset 'READWRITE'

NB. get 1000 objects in each get loop pass
dpset 'GETFACTOR';1000
```

2.10 ed – edit dictionary objects

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

Monad: *ed clName ∨ blclNames*

```
NB. retrieve word and place in edit window
ed 'word'

NB. put many words in edit window
ed ;:'many words edited'
```

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

```
NB. edit test
1 ed 'test'

NB. generate group and place in edit window
2 ed 'group'

NB. generate test suite and place in edit window
3 ed 'suite'

NB. edit macro text
4 ed 'macro'

NB. edit group header text
```

```
2 1 ed 'group'

NB. edit suite header text
3 1 ed 'suite'
```

2.11 et – put text into edit window

et load character lists into edit windows.

Monad: *et clText*

```
NB. put character data into edit window
et 'put text in edit window'

NB. read text and put in edit window
et read 'c:\temp\text.txt'
```

2.12 get — get objects

get retrieves dictionary objects and information about dictionary objects. There is a close correspondence between the arguments of get and put, see subsection 2.27, on page 28. A basic JOD rule is that if you can put it you can get it.

Monad: *get clName* $\vee$ *blclNames*

```
NB. get word and define in current locale
get 'word'

NB. get a group
get }. grp ''
```

Dyad: *ilOptions get clName* $\vee$ *blclNames*
clLocale get clName $\vee$ *blclNames*

```
NB. get word (monad)
0 get 'word'

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

```
NB. for words a character left
NB. argument is a target locale

NB. get into locale
'locale' get ;:'hi ho into locale we go'

NB. allow numbered locales
'666' get ;:'beast code'

NB. explain words
0 8 get ;:'explain us ehh'

NB. word documentation
0 9 get ;:'document or die'

NB. get word scripts without defining
0 10 get 'define';'not'

NB. information about stored
NB. words can be retrieved with get

NB. J name class of words
0 12 get ;:'our name class'

NB. word creation dates
0 13 get ;:'our creation'

NB. last word put dates
0 14 get ;:'last change'

NB. word size in bytes
0 15 get ;:'how big are we'

NB. get test scripts
1 7 get 'i';'test';'it'

NB. test explanations
1 8 get ;:'explain tests'

NB. test case documentation
1 9 get 'radical'

NB. information about stored tests
NB. test creation dates
1 13 get ;:'our creation'

NB. last test put dates
1 14 get ;:'last change'

NB. test size in bytes
1 15 get ;:'how big are we'

NB. get group scripts
2 7 get ;:'groupies cool'

NB. get group explaination text
2 8 get 'group';'explain'

NB. get group document text
2 9 get 'document'

NB. suite text
3 7 get ;:'this suites me'

NB. explain suites
3 8 get ;:'suites need comments'

NB. document suites
3 9 get ;:'document your suites'

NB. get various macros
4 get 'jmacro';'html';'latex'

NB. explain macros
4 8 get ;:'macros need explaining'
4 9 get ;:'and documents too'
```

2.13 `getrx` – get required to execute

`getrx` gets all the words required to execute words on `(y)`.

Warning: if the words listed on `(y)` refer to object or locale references this verb returns an error because such words generally cannot be run out of context.

Monad: *getrx clName* $\vee$ *blclNames*

```
NB. load required words into base locale
getrx 'stuffineed'

NB. get all words required to run many words
getrx ;:'stuff we need to run'
```

Dyad: *clLocale getrx clName ∨ blclNames*

```
NB. load all required words into locale
'locale' getrx ;:'load the stuff we need into locale'
```

2.14 `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 swept under the proverbial rug.

For example the `jfiles` utility is often accessed with `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*

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

Dyad: *iaObject globs clName*

```
NB. update referenced globals
0 globs 'word'

NB. update all words in a group
0 globs&> }. grp 'group'

NB. list global references in test text
1 globs 'test'

NB. classify name references in locale word.
11 globs 'word'
```

2.15 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 dictionarys 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*

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

NB. list words in group
grp 'group'

NB. create/reset groupfirst 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*

```
NB. list all test groups (suites) (3 dnl '')
3 grp ''

NB. list tests in suite
3 grp 'suite'

NB. (monad)
2 grp 'group';'list';'of';'group';'names'

NB. create/reset suite
3 grp 'suite';'list';'of';'test';'names'

NB. empty suite
3 grp <'suite'
```

2.16 `gt` – get edit window text

Fetch text from edit window.

Monad: *gt zl ∨ clName*

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

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

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

2.17 `hlpnl` – display short object descriptions

`hlpnl` displays short object descriptions.

Short object descriptions are always a good idea. If you cannot *tersely* describe an object you probably don't understand it. Short descriptions are stored with `put`.

Monad: *hlpnl clName ∨ blclNames*

```
NB. put short word description
0 8 put 'describeme';'briefly describe me'

NB. display short word description
hlpnl 'describeme'

NB. display many descriptions
hlpnl ;:'show our short word descriptions'

NB. describe all the words in a group
hlpnl }. grp 'groupname'

NB. describe all the words called by a word
hlpnl allrefs <'wordname'

NB. describe all dictionary words
hlpnl }. dnl ''
```

Dyad: *iaObject hlpnl clName* ∨ *blclNames*

```
NB. display short word description (monad)
0 hlpnl 'word'

NB. display test, group, suite, macro descriptions
1 hlpnl 'testname'
2 hlpnl 'groupname'
3 hlpnl 'suitename'
4 hlpnl 'macroname'

NB. describe a test suite
3 hlpnl }. 3 dnl 'testsuite'

NB. describe a group
2 hlpnl }. 2 dnl 'groupname'

NB. describe macro scripts with prefix 'prj'
4 hlpnl }. 4 dnl 'prj'
```

2.18 `jodage` — age of JOD objects

`jodage` returns the age of JOD objects. When an object is put into a dictionary the date is recorded.

The monad returns the age of words and the dyad returns the age of other objects. JOD dates are stored in a fractional day `yyyymmdd.fd` floating point format.[2]

Monad: *`jodage clWord ∨ blclWords`*

```
NB. show age of (jodage)
jodage 'jodage'

NB. age of all group words
jodage }. grp 'bstats'
```

Dyad: *`ia jodage clWord ∨ blclNames`*

```
NB. age of all test scripts
1 jodage }. 1 dnl ''

NB. age of group script
2 jodage 'mygroup'

NB. age of all macro scripts
4 jodage }. 4 dnl ''
```

2.19 `jodhelp` — return help

`jodhelp` displays online help for JOD words. The monad returns help for specific words and displays an index. The dyad lists all words that have help.

Monad: *`jodhelp clWord`*

```
NB. show (put) help
jodhelp 'put'

NB. display help index
jodhelp ''
```

[2] JOD times are derived from *local* computer clock times. `UTC` is not used.

Dyad:*uuIgnore jodhelp uuIgnore*

```
NB. list help topics - ignores arguments
jodhelp~ 0
```

2.20 `lg` – make and load group

`lg` assembles and loads JOD group scripts. The monad loads without the postprocessor script and the dyad loads with the postprocessor.

The postprocessor is a JOD macro script that is associated with a group. If a group is named `numtutils` the associated postprocessor is named `POST_numutils`. The prefix `POST_` labels macro scripts as postprocessors. The postprocessor is appended to generated group scripts and is often used to start systems.

Monad: *lg clGroup*

```
NB. make and load group without postprocessor
lg 'groupname'
```

Dyad: *iaOption lg clGroup*

```
NB. monad
2 lg 'groupname'

NB. define a group postprocessor macro script
NB. 21 identifies macro text as an arbitrary J script
4 put 'POST_groupname';21;'smoutput ''hello world'''

NB. make and load appending postprocessor
lg~ 'groupname'
```

2.21 `locgrp` – list groups/suites with word/test

`locgrp` lists groups and suites with word or test `(y)`. A word or test can belong to many groups or suites.

Monad: *logrp clName*

```
NB. list all groups that contain 'myword'
locgrp 'myword'

NB. list all suites that contain 'thistest'
locgrp 'thistest'
```

2.22 `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: see Appendix **??** Generated Script Structure, on page **??**.

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 serial 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
NB. to put dictionary dump directory.
NB. The name of the put dictionary is
NB. used as the dump file name.
make ''

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

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'

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

NB. make J script that defines a suite
3 make 'suite'
```

An option code controls whether results are written to file, (1 default), or returned, (2 return), for word lists, groups and suites. Default dictionary file locations are the subdirectories created by `newd`.

```
NB. make and return group script
2 2 make 'group'

NB. make put dictionary suite script and write to file
3 1 make 'suite'

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'

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

2.23 `mls` — make load script

`mls` generates J load scripts. The generated script is added to:

```
~system\extras\config\scripts.ijs
```

and can be loaded with the standard J load utility. The load script is independent of JOD and can be used like any other J load script.

The generated script can be written to file or returned. Generated scripts are stored in the put dictionary script subdirectory. `mls` appends any postprocessor to the generated script: see Appendix **??** Generated Script Structure, on page **??**.

Monad: *mls clGroupname*

```
NB. add a postprocessor script for (addgroup)
postproc=. 'smoutput ''''this is a post processor'''''
4 put 'POST_appgroup';JSCRIPT_ajod_;postproc

NB. generate group script with
NB. postprocessor and add to scripts.ijs
mls 'appgroup'

NB. load group - postprocessor runs
load 'appgroup'
```

Dyad: *iaOption mls clGroupname*

```
NB. make J script file but do
NB. not add to scripts.ijs
0 mls 'bstats'

NB. monad
1 mls 'bstats'

NB. return generated script as result
NB.  does not add to scripts.ijs
2 mls 'bstats'
```

2.24 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
NB. is created in the default directory
newd 'makemydictionary'

NB. create with name in location
newd 'new';'c:\location\'

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

2.25 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 puts the first dictionary is called the *put dictionary*.

Monad: *od zl ∨ clDictionary ∨ blclDictionaries*

```
NB. list registered dictionaries
od ''

NB. open read/write
od 'dictionary'

NB. opens di read/write
od 'd1';'d2';'d3'
```

Dyad: *iaOption od zl ∨ clDictionary*
iaOption od zl blclDictionaries

```
NB. list registered dictionaries (monad)
1 od ''

NB. close all open dictionaries (related to did 4)
3 od ''

NB. open read/write (monad)
1 od 'dictionary'

NB. open read only and append to any path
2 od 'dictionary'

NB. open di read only and append to any path
2 od 'd1';'d2';'d3'

NB. close dictionaries and remove from path
3 od ;:'d0 d1 d2'

NB. all dictionary root directories
4 od ''

NB. list all dictionaries as regd script
5 od ''
```

2.26 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*

```
NB. packd requires an open READWRITE dictionary
od 'dictionary'

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

2.27 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*

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

NB. store all base locale verbs in dictionary
put nl 3
```

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

```
NB. put words (monad)
0 put ;:'w0 w1 w2 w3 w4'

NB. put words from specified locale
'locale' put 'w0';'w2';'w3'

NB. numbered locales
'99' put 'word'

NB. put explain/document text
NB. 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 header scripts from name,value table
NB. A group header script is an arbitrary J script
NB. that preceeds the code generated by make.

NB. Group header scripts can be put
NB. with 2 1 as well - maintains put/get symmetry

2 put (;:'g0 g1'),.('text ...';'text ...')
2 1 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 header scripts from name value table
3 put (;:'s0 s1'),.('text ...';'text ...')
3 1 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

NB. J scripts - can be run with (rm)
4 put (;:'m0 m1'),.(21;21),.('text ...';'...')

NB. LaTeX
4 put (;:'m0 m1'),.(22;22),.('text ...';'...')

NB. HTML
4 put (;:'m0 m1'),.(23;23),.('text ...';'...')

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

NB. plain ASCII text
4 put (;:'m0 m1'),.(25;25),.('text ...';'...')

NB. UTF-8 unicode text
4 put (;:'m0 m1'),.(26;26),.('text ...';'...')

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

2.28 `regd` — register dictionaries

`restd` 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 JOD 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
NB. directory and dictionary must exist
regd 'name';'c:\location\'

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

Dyad: *iaOption regd clDictionary*

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

NB. regd can be used to rename dictionaries
NB. and update dictionary documentation

NB. unregister
'name path' =. _2 {. 3 regd 'badname'

NB. re-register with new name and documentation
doc =. 'brand spanking new documenation'
regd 'goodname';path;doc
```

2.29 restd – restore backup dictionaries

restd restores the last backup created by packd.

Monad: *restd clDictionary*

```
NB. open dictionary READWRITE
NB. must be first dictionary on the path
od 'lastbackup' [ 3 od ''

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

2.30 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*

```
NB. all put dictionary words in last put order
revo ''

NB. revised words with names beginning with 'boo'
revo 'boo'
```

Dyad: *iaObject revo zl ∨ clName*

```
NB. list all revised tests
1 revo ''

NB. revised suites with names prefixed by 'boo'
3 revo 'boo'
```

2.31 `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*

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

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

Dyad: *iaOption rm zl ∨ clName ∨ blclNames*

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

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

2.32 `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`*

```
NB. run test script as a tautology
rtt 'tautologytest'

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

Dyad: *`iaOption rtt clName ∨ blclNames`*

```
NB. same as monad
0 rtt 'tautologytest'

NB. run tautology test and suppress output
1 rtt 'silenttautology'

NB. run test as plain script
2 rtt 'plaintest'

NB. generate test suite and run as tautology
3 rtt 'suitename'

NB. generate test suite and run as silent tautology
4 rtt 'silentsuite'
```

2.33 `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. Its important to know an object is being referenced even if you cannot peer inside the object.

Monad: *uses blclName ∨ clName*

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

Dyad: *iaObject uses blclName ∨ clname*

```
NB. same as monad
0 uses 'word'

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

3 JOD Directory and File Layouts

3.1 Master File — `jmaster.ijf`

`jmaster.ijs` is a binary component `jfile`. To use `jfiles` you load or require the standard `jfiles` script.

`jmaster.ijf` is an index of currently registered dictionaries and standard dictionary metadata. The component layout of `jmaster.ijf` is given in Table 2 on page 35.

Component	Hungarian	Description
c_0	`(pa;il)`	Use bit and last master change. The use bit is set by all processes that update this file - while set the use bit blocks other dictionary tasks from updating this file.
c_1	`(cl;i,xi)`	Version `m.m.p` character, build count and unique master file id.
c_2	`bt`	Dictionary names, numbers, directories and read-write status. When a dictionary is opened for update (`READWRITE` default) by `od` the status is set and stays on until closed by `od`. This blocks all other dictionary tasks from using the dictionary. This harsh treatment prevents garbled files. Dictionaries can also be opened read only. This allows multiple readers but no writers.
c_3	`bt`	Previous master directory. Essentially a copy of component two less at most one deleted or new dictionary.
$c_4 \rightarrow c_6$		Reserved.
c_7	`bt`	Active dictionary parameters. `0 { - blcl` ; parameter names `1 { - blcl` ; short parameter explanation `2 { - bluu` ; default values
c_8	`bt`	Copy of active dictionary parameters.
c_9	`bt`	Default dictionary parameters.
c_{10}	`xil`	Dictionary log. The dictionary log is a simple, (append only), list of all the extended dictionary numbers that have ever been registered. When a dictionary is registered it is appended to this list. If it is unregistered and then re-registered the same dictionary number will appear more than once. I don't expect this list to be very large. Hundreds, maybe thousands, over the lifetime of the master file.

Table 2: `jmaster.ijf` file component layout

3.2 Words File — `jwords.ijf`

`jwords.ijs` is a binary component `jfile`.

`jwords.ijf` contains word definitions and metadata. The component layout of `jwords.ijf` is given in Table 3 on page 36.

3.3 Tests File — `jtests.ijf`

`jtests.ijs` is a binary component `jfile`.

Component	**Hungarian**	**Description**
c_0	`blnl`	Length and last directory change.
c_1	`il`	Pack and backup count. Used to prefix backup and dump files.
c_2	`blcl`	Dictionary documentation `newd`, `regd`.
c_3	`bluu`	Dictionary parameters. `0  { - cl` ; dictionary name `1  { - ra` ; dictionary number (extended precision) `2  { - il` ; dictionary creation date `3  { - il` ; last dump date (NOT UPDATED) `4  { - cl` ; script directory `5  { - cl` ; suite directory `6  { - cl` ; macro directory `7  { - cl` ; document directory `8  { - cl` ; dump directory `9  { - cl` ; alien directory `10 { - cl` ; J version that created dictionary `11 { - ia` ; J system code that created dictionary `12 { - uu` ; unused - reserved `13 { - bt` ; user dictionary parameters see: `jmaster.ijf`. `0 { cl` ; parameter `1 { uu` ; value
		Main inverted items, $c_4 \longrightarrow c_{11}$ have the same length.
c_4	`blcl`	Word list (main index 1).
c_5	`il`	Word components (main index 2).
c_6	`il`	Name class list.
c_7	`fl`	Last put date list `yyyymmdd.fd` (fractional day).
c_8	`fl`	Creation put list `yyyymmdd.fd` (fractional day).
c_9	`il`	Word size in bytes.
c_{10}		Reserved.
c_{11}	`blcl`	Short word explanations.
$c_{12} \longrightarrow c_{38}$		Reserved.
		The remaining component pairs contain word data. The word names match the entries in the word index list.
c_{39}	`bluu`	Word definition. `0 { - cl` ; word name `1 { - ia` ; name class `2 { - cl` $\vee$ `uu` ; word value, nouns are stored in binary all other words are character lists
c_{40}	`bluu`	Word documentation and other. `0 { - cl` ; word name `1 { - uu` ; unused - reserved `2 { - uu` ; unused - reserved `3 { - cl` ; text documentation
c_{41}	`bluu`	Like c_{39}
c_{42}	`bluu`	Like c_{40}
...	...	...
c_n	...	Like c_{n-2}

Table 3: `jwords.ijf` file component layout

`jtests.ijf` contains test definitions and metadata. The component layout of `jtests.ijf` is given in Table 4 on page 37.

Component	Hungarian	Description
c_0	`blnl`	Length and last directory change.
$c_1 \rightarrow c_3$		Reserved.
		Main inverted items, $c_4 \rightarrow c_{11}$ have the same length.
c_4	`blcl`	Test list (main index 1).
c_5	`il`	Test components (main index 2).
c_6		Reserved to match `jwords.ijf`.
c_7	`fl`	Last put date list `yyyymmdd.fd` (fractional day).
c_8	`fl`	Creation put list `yyyymmdd.fd` (fractional day).
c_9	`il`	Test size in bytes.
c_{10}		Reserved.
c_{11}	`blcl`	Short test explanations.
$c_{12} \rightarrow c_{38}$		Reserved.
		The remaining component pairs contain test data. The test names match the entries in the test index c_4 list.
c_{39}	`blcl`	Test definition. `0 { - cl` ; test name `1 { - cl` ; test value
c_{40}	`bluu`	Test documentation and other. `0 { - cl` ; test name `1 { - uu` ; unused - reserved `2 { - uu` ; unused - reserved `3 { - cl` ; text documentation
c_{41}	`blcl`	Like c_{39}
c_{42}	`bluu`	Like c_{40}
...	...	...
c_n	...	Like c_{n-2}

Table 4: `jtests.ijf` file component layout

3.4 Groups File — `jgroups.ijf`

`jgroups.ijs` is a binary component `jfile`.

`jgroups.ijf` contains group definitions and group metadata. The component layout of `jgroups.ijf` is given in Table 5 on page 38.

3.5 Suites File — `jsuites.ijf`

`jsuites.ijs` is a binary component `jfile`.

`jsuites.ijf` contains test suite definitions and test suite metadata. The component layout of `jsuites.ijf` is given in Table 6 on page 39.

Component	Hungarian	Description
c_0	`blnl`	Group count and last directory change.
$c_1 \longrightarrow c_3$		Reserved.
		Main inverted items, $c_4 \longrightarrow c_{11}$ have the same length.
c_4	`blcl`	Group list (main index 1).
c_5	`il`	Group components (main index 2).
c_6		Reserved to match `jwords.ijf`.
c_7	`fl`	Last put date list `yyyymmdd.fd` (fractional day).
c_8	`fl`	Creation put list `yyyymmdd.fd` (fractional day).
$c_9 \longrightarrow c_{10}$		Reserved.
c_{11}	`blcl`	Short group explanations.
$c_{12} \longrightarrow c_{38}$		Reserved.
		The remaining component pairs contain group data. The group names match the entries in the group index c_4 list.
c_{39}	`bluu`	Group definition. `0 { - cl`; group name `1 { - cl`; group prefix script `2 { - blcl`; group content list
c_{40}	`bluu`	Group documentation and other. `0 { - cl`; group name `1 { - uu`; unused - reserved `2 { - uu`; unused - reserved `3 { - cl`; text documentation
c_{41}	`bluu`	Like c_{39}
c_{42}	`bluu`	Like c_{40}
…	…	…
c_n	…	Like c_{n-2}

Table 5: `jgroups.ijf` file component layout

Component	**Hungarian**	**Description**
c_0	`blnl`	Suite count and last directory change.
$c_1 \longrightarrow c_3$		Reserved.
		Main inverted items, $c_4 \longrightarrow c_{11}$ have the same length.
c_4	`blcl`	Suite list (main index 1).
c_5	`il`	Suite components (main index 2).
c_6		Reserved to match `jwords.ijf`.
c_7	`fl`	Last put date list `yyyymmdd.fd` (fractional day).
c_8	`fl`	Creation put list `yyyymmdd.fd` (fractional day).
$c_9 \longrightarrow c_{10}$		Reserved.
c_{11}	`blcl`	Short suite explanations.
$c_{12} \longrightarrow c_{38}$		Reserved.
		The remaining component pairs contain suite data. The suite names match the entries in the suite index c_4 list.
c_{39}	`bluu`	Suite definition. `0 { - cl` ; suite name `1 { - cl` ; suite prefix script `2 { - blcl` ; suite content list
c_{40}	`bluu`	Suite documentation and other. `0 { - cl` ; suite name `1 { - uu` ; unused - reserved `2 { - uu` ; unused - reserved `3 { - cl` ; text documentation
c_{41}	`bluu`	Like c_{39}
c_{42}	`bluu`	Like c_{40}
…	…	…
c_n	…	Like c_{n-2}

Table 6: `jsuites.ijf` file component layout

3.6 Macros File — `jmacros.ijf`

`jmacros.ijs` is a binary component `jfile`.

`jmacros.ijf` contains macro script definitions and macro script metadata. The component layout of `jmacros.ijf` is given in Table 7 on page 40.

Component	Hungarian	Description
c_0	`blnl`	Macro count and last directory change.
$c_1 \rightarrow c_3$		Reserved.
		Main inverted items, $c_4 \rightarrow c_{11}$ have the same length.
c_4	`blcl`	Macro list (main index 1).
c_5	`il`	Macro components (main index 2).
c_6		Reserved to match `jwords.ijf`.
c_7	`fl`	Last put date list `yyyymmdd.fd` (fractional day).
c_8	`fl`	Creation put list `yyyymmdd.fd` (fractional day).
c_9	`fl`	Macro size in bytes.
c_{10}		Reserved.
c_{11}	`blcl`	Short macro explanations.
$c_{12} \rightarrow c_{38}$		Reserved.
		The remaining component pairs contain macro data. The macro names match the entries in the macro index c_4 list.
c_{39}	`blcl`	Macro definition. `0 { - cl` ; macro name `1 { - cl` ; macro script
c_{40}	`bluu`	Macro documentation and other. `0 { - cl` ; macro name `1 { - uu` ; unused - reserved `2 { - uu` ; unused - reserved `3 { - cl` ; text documentation
c_{41}	`blcl`	Like c_{39}
c_{42}	`bluu`	Like c_{40}
...	...	...
c_n	...	Like c_{n-2}

Table 7: `jmacros.ijf` file component layout

3.7 Uses File — `juses.ijf`

`juses.ijs` is a binary component `jfile`.

`juses.ijf` contains word references: see `globs` subsection 2.14, on page 18.

4 JOD Distribution

JOD is distributed as a *J addon*. You can instal JOD using the *J package manager*.

The JOD distribution is broken into two packages:

Component	Hungarian	Description
c_0	`blnl`	0 and and last directory change. The number of references stored is not tracked. 0 is the value in the count position of other files.
$c_1 \longrightarrow c_4$		Reserved.
		Uses (reference) directory layout differs from `jwords.ijf` but occupies the same component range for `packd`. Only non-empty reference lists are stored.
c_5	`blcl`	Word uses words (index).
c_6	`il`	Component list.
$c_7 \longrightarrow c_{18}$		Reserved.
c_{19}	`xil`	Put reference path. List of extended dictionary numbers `DIDNUMS`.
$c_{20} \longrightarrow c_{38}$		Reserved.
		Note: remaining components contain reference lists where: `cl` is the name of the object being referenced. `ia` is an object code - 0 means words used by words. `(<blcl),<blcl` is a pair of boxed lists. The first list contains all global references excluding locale references. Locale references, if any, are in the second list.
c_{39}	`cl;ia;(<blcl),<blcl`	References.
c_{40}		Like c_{39}
...	...	...
c_n	...	Like c_{n-1}

Table 8: `juses.ijf` file component layout

1. `jod`: This is the only package that must be installed to run JOD. It contains JOD system code, documentation and other supporting files.

2. `jodsource`: This addon is single `zip` file containing three serialized JOD dictionary dumps. JOD dictionary dumps are J script files that can rebuild JOD dictionaries. Dump files are the best way to distribute dictionary code since they are independent of J binary representations. The `jodsource` addon contains.

   (a) `joddev.ijs` — development put dictionary

   (b) `jod.ijs` — main JOD source and documentation

   (c) `utils.ijs` — common utilities

List of Tables

1 Version History 1
2 `jmaster.ijf` file component layout 35
3 `jwords.ijf` file component layout 36
4 `jtests.ijf` file component layout 37
5 `jgroups.ijf` file component layout 38
6 `jsuites.ijf` file component layout 39
7 `jmacros.ijf` file component layout 40
8 `juses.ijf` file component layout 41

List of Figures

1 JOD Classes 4