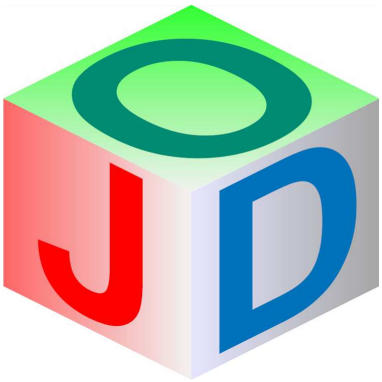


J Object Dictionary

A code, test and documentation database system for J

Author:
John D. Baker
bakerjd99@gmail.com

Release:
0.9.97
March 25, 2015

Print dimensions: US Letter
Bound printed version at www.lulu.com
Printed Version Link

Document Version History		
Date	**Version**	**Description**
March 25, 2015	0.9.97	current version
June 14, 2014	0.9.94	prior JAL release
November 15, 2012	0.9.85	JAL update
April 21, 2012	0.9.75	Lulu draft
December 9, 2011	0.9.6	Lulu draft
November 30, 2011	0.9.5	changed JOD logo
October 31, 2008	0.8.0	first printed edition
October 7, 2008	0.7.3	first β edition
September 24, 2008	0.7.1	release draft
August 18, 2008	0.6.0	release draft
...	...	...
March 30, 2008	0.3.5	first draft

Table 1: Abbreviated Document Version History

Contents

1 Introduction

1.1 What is JOD?

JOD is a code, test and documentation database *Addon* for the *J programming language* [3]. JOD has been programmed entirely in J[1] and can be quickly ported to any system that supports J.

1.2 Why JOD?

Programming in J has a charming and distinctive flavor. Tasks decompose into scores of tiny programs that are called *words*. JOD stores and organizes J words and other objects in a dictionary database: hence the name **J O**bject **D**ictionary.

Code 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 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? Scattering them in many scripts leads to error prone *copy-and-pasting* or *rampant over-inclusion*.[2] The best way to reuse short words 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 found in many files it's not always easy to find the current version.

- With JOD there are no significant limits on vocabulary size. Scripts *can* hold thousands of words but it's a nuisance to maintain and include such large files.

- The *complete definition* of a word 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 will find descriptions of basic algorithms, remarks about coding techniques, references to published material, program test suites, detailed error logs and germane diagrams. Storing such material in source code would horribly clutter programs. A dictionary is where this material belongs.

- *Relationships between words* can be stored. Accurate word references make it easier to understand code. This is especially true if references and documentation are linked.

[1] JOD makes a few OS calls to move files and generate `GUIDs`.

[2] 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.

- JOD facilitates the *generation of scripts and the distribution* of code. When I program with JOD I rarely write `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.

- Finally, *JOD encourages a different way to think about programming.* When programs are reduced to their primary reusable units, words in J's case, many traditional software engineering problems almost disappear. With JOD code reuse and refactoring is fluid, natural and darn near unavoidable.

2 Installing and Configuring JOD

Before using JOD you need to install the current Windows, Linux or Mac version of J.[3] J can be downloaded from `www.jsoftware.com`.

In addition to J you need to install a number of J *addons*. JOD uses the `jfiles`, `regex` and `task` addons. Some J versions include these addons as part of the basic J system. If you can run the following J command without errors your system is ready for JOD.

```
require 'jfiles regex task'
```

If you encounter any errors use `JAL` to install missing addons.

JOD can be installed in two ways.

1. Use `JAL`, J's package manager [10], to download JOD. *Using JAL is the easiest way to install and maintain JOD.*

2. Download the current JOD distribution from *The JOD Page* [6] and unzip, preserving directories, to the relative directory[4] `~addons/general/jod`

After installation JOD can be loaded with:[5]

```
load 'general/jod'
```

To configure and maintain JOD you must be aware of the following:

1. JOD uses the J startup file `~config/startup.ijs` to store load scripts: see `mls` on page 38. Exercise caution when manually editing JOD's load script section.

[3] JOD runs on J `6.0x` , J `7.0x` and J `8.0x` Windows systems. The Linux and Mac versions of JOD require J `7.0x` or `8.0x`. Currently there are no `IOS` or `Android` versions of JOD. Porting JOD to any J system is a simple task. If you are interested in helping me create ports please contact me at: `bakerjd99@gmail.com`

[4] Paths that begin with the `~` character are relative directories. The full path can be obtained with `jpath` verb.

[5] The `'` character is a single quote: in J notation it is: `39 { a.`

2. To run JOD labs you must download and install the `jodsource` addon [5]. `jodsource` can be installed with `JAL`.

3. JOD labs and test scripts assume some J folders have been configured. Open J's configuration tool, see Figure 1 on page 7, and define folders like:

```
JOD                c:/jod
JODDUMPS           c:/jod/joddumps
JODSOURCE          c:/jodtest/labtesting
JODTEST            c:/jodtest/test
```

   Use fully qualified directory paths that are not in the J install tree.

4. `jodhelp` (35) uses a PDF reader to display JOD documentation. Use J's configuration tool to set your preferred PDF reader.[6]

3 Quick Tutorial

The best way to get started with JOD is to work through the lab *JOD (1) Introduction*.[7] JOD labs are listed in the `General` category: see Figure 2 on page 8. If JOD labs are not listed in the `General` category browse the directory:

```
~addons/general/jod/jodlabs
```

This tutorial uses lab material; work through the JOD labs after reading this section.

Start JOD. After installation JOD can be started from a J session with:

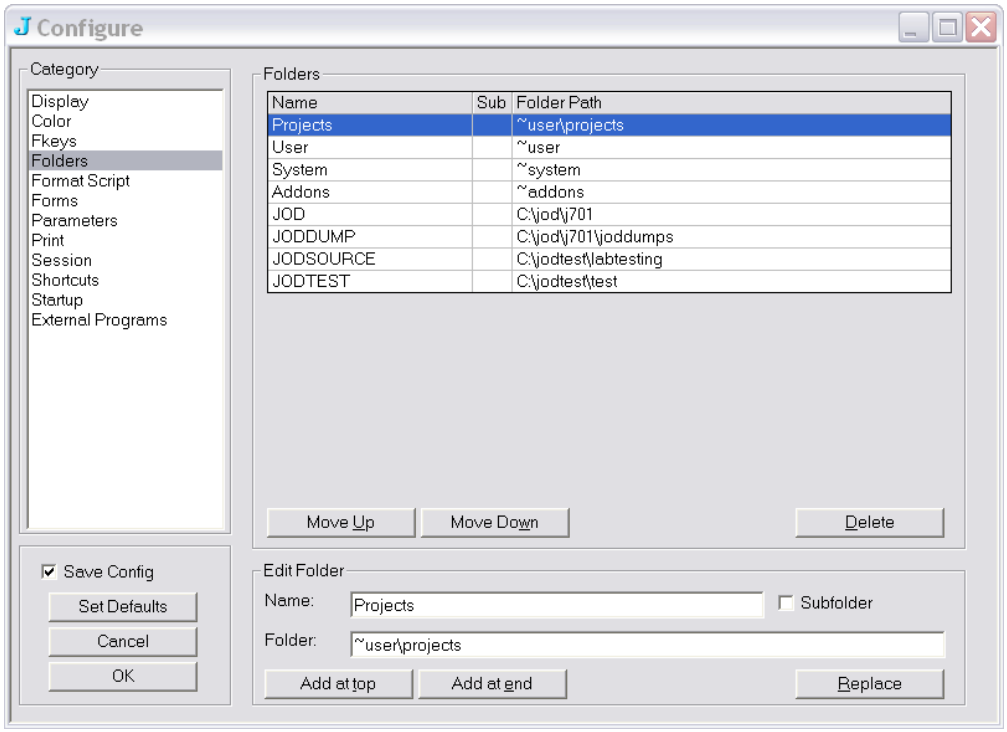
```
require 'general/jod'   NB. start JOD
```

Create dictionaries. To use JOD you must create some dictionaries with `newd`: see page 39. JOD words, with a few exceptions, return boxed list results. The first item is a return code where `1` indicates success and `0` means failure.

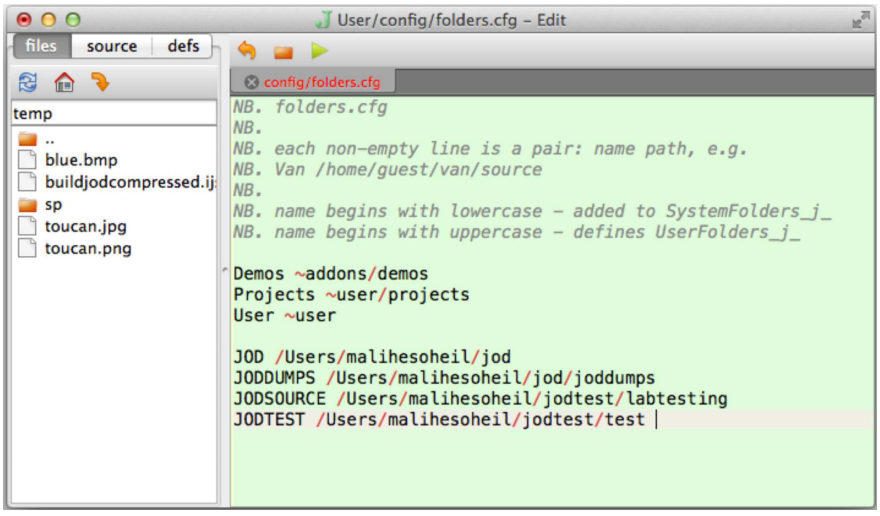
```
   newd 'lab';'c:/jodtut/lab'   NB. create (lab)
+-+--------------------+---+--------------+
|1|dictionary created ->|lab|c:/jodtut/lab/|
+-+--------------------+---+--------------+

   newd 'labdev';'c:/jodtut/labdev'  NB. create (labdev)
```

[6] See the blog post JOD Update: Version `0.9.97*` for J configuration advice.
[7] The number "(*n*)" in JOD lab titles is a suggested order.

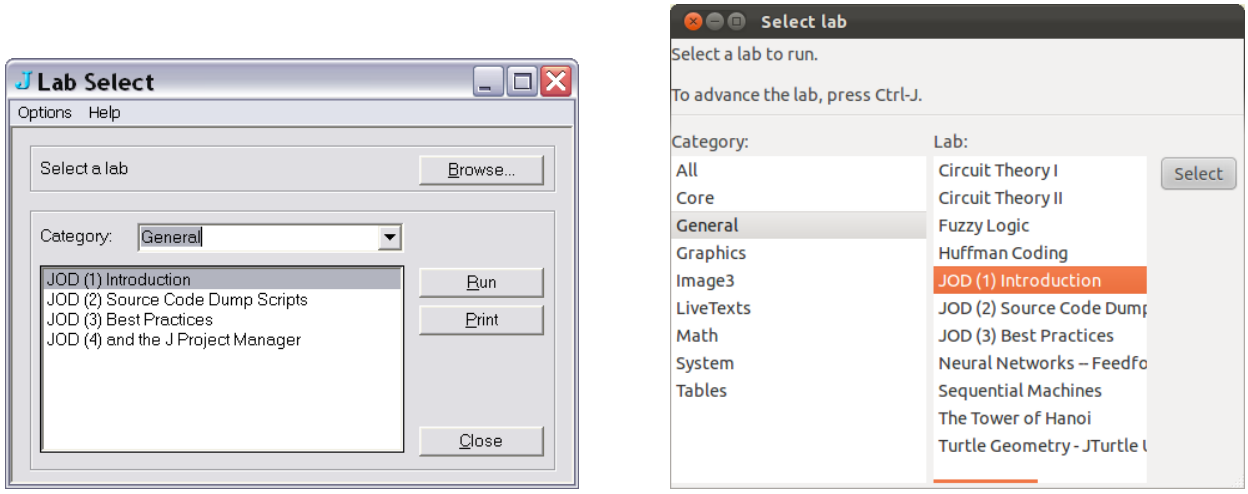


(a) J 6.0x Windows configuration dialog



(b) J 8.0x Mac JQT configuration editor

Figure 1: This folder configuration is recommended for running JOD labs and test scripts. When defining JOD folders the full path, including the drive letter on Windows and a leading / for Linux and the Mac must be used: see page 6.



(a) J `6.0x` Windows (b) J `7.0x` JGTK Linux

Figure 2: JOD Labs are installed in the `General` category.

```
+-+---------------------+------+-----------------+
|1|dictionary created ->|labdev|c:/jodtut/labdev/|
+-+---------------------+------+-----------------+
```

Open dictionaries. To use a dictionary it must be open. Open dictionaries with `od`: see page 43.

```
   od 'labdev'    NB. open read/write
+-+--------------+------+
|1|opened (rw) ->|labdev|
+-+--------------+------+

   2 od 'lab'     NB. open read/only
+-+--------------+---+
|1|opened (ro) ->|lab|
+-+--------------+---+
```

Show open dictionaries. Open dictionaries define a *reference path*, see appendix C on page 63. `did`, see page 20, displays information about open dictionaries.

```
   did 0  NB. show open dictionary path
+-+------+---+
|1|labdev|lab|
+-+------+---+
```

Create some words to store. You can store all types of J words in JOD dictionaries.

```
NB. create some words in the base locale
random=: ?10 10$100  NB. numeric noun
text=: 'this is a test of the one pure thing'
floats=: 2 + % 100#100
symbols=: s: ' once more with feeling'
boxed=: <"1 i. 2 3
rationals=: 100 + % (>:i. 10x) ^ 50
unicode=: u: 'this is now unicode'
each=: &.>           NB. tacit adverb

explicit=: 4 : 0
NB. explicit verb
x +. y
)

NB. list of defined words
words=: ;:'random text floats symbols'
words=: words, ;:'boxed rationals unicode each explicit'
```

Store words in put dictionary. The first dictionary on the path is the only dictionary that can be updated. Most updates are put operations so the first dictionary is called the *put dictionary*: see put on page 44.

```
   put words    NB. save words
+-+------------------+------+
|1|9 word(s) put in ->|labdev|
+-+------------------+------+

   erase words  NB. erase words
 1 1 1 1 1 1 1 1 1
```

Retrieve words from dictionaries. get, see page 28, fetches words from dictionaries.

```
   get words     NB. get words
+-+-----------------+
|1|9 word(s) defined|
+-+-----------------+
```

Make a group. Dictionary words can be grouped: see grp page 32.

```
grp  'tutgroup' ; words
```

```
+-+-------------------------+------+
|1|group <tutgroup> put in ->|labdev|
+-+-------------------------+------+
```

Make a load script from a group. Load scripts are J scripts that can be loaded with the standard `load` utility. Standard J load scripts are defined in the `scripts.ijs` file. This file is reset by `JAL` updates so JOD load scripts are stored in the user's `startup.ijs` file: see appendix H on page 70.

```
   mls 'tutgroup'
+-+-------------------+----------------------------------+
|1|load script saved ->|c:/jodtut/labdev/script/tutgroup.ijs|
+-+-------------------+----------------------------------+

   NB. load with standard utility
   load 'tutgroup'
```

Backup the put dictionary. *You're either backed up or f'ed up—there are no other options!* JOD makes backups easy: see `packd` on page 44.

```
   packd 'labdev'
+-+-------------------+------+-+
|1|dictionary packed ->|labdev|1|
+-+-------------------+------+-+
```

Dump dictionaries on path. `make`, see page 37, can dump all open dictionaries as a single *dump script.*

```
   make ''
+-+-------------------------+-------------------------------+
|1|object(s) on path dumped ->|c:/jodtut/labdev/dump/labdev.ijs|
+-+-------------------------+-------------------------------+

   3 od ''      NB. close all dictionaries
+-+---------+------+---+
|1|closed ->|labdev|lab|
+-+---------+------+---+
```

This brief tutorial has just touched JOD's surface. Work through the JOD labs to learn more.

4 Best Practices

Here are some JOD practices I have found useful. A JOD lab, *JOD (3) Best Practices*, elaborates and reiterates this material. After reading these notes I recommend you run this lab. JOD labs are found in the `General` lab category.

JOD does not belong in the J tree. Never store your JOD dictionaries in J install directories! Create a JOD master dictionary directory root that is independent of J: see `newd` on page 39. It's also a good idea to define a subdirectory structure that mirrors J's versions.

```
NB. create a master JOD directory root outside of J's directories.
newd 'bptest';'c:/jodlabs/j701/bptest';'best practices dictionary'

NB. linux and mac paths start with a leading /
newd 'bptest';'/home/john/jodlabs/bptest'
```

Backup backup and then backup. It's easy to backup with JOD so backup often: see `packd` and `restd` on pages 44 and 48.

```
NB. open the best practice dictionary
od 'bptest' [ 3 od ''

NB. back it up
packd 'bptest'
```

Take a script dump. It's a good idea to "dump" your dictionaries as plain text. JOD can dump all open dictionaries as a single J script: see `make` on page 37. Script dumps are the most stable way to store J dictionaries. The `jodsource` addon distributes all JOD source code in this form.

```
NB. (make) creates a dictionary dump in the dump subdirectory
make ''
```

Make a master re-register script. JOD only sees dictionaries registered in the `jmaster.ijf` file: see Table 3 on page 54. Maintaining a list of registered dictionaries is recommended. JOD can generate a re-register script: see `od` on page 43. Generate a re-register script and put it in your main JOD dictionary directory root.

```
NB. generate re-register script
rereg=:  ;{: 5 od ''
```

Set library dictionaries to `READONLY`. Open JOD dictionaries define a search path. The first dictionary on the path is the only dictionary that can be changed. It is called the *put dictionary.* Even though nonput dictionaries cannot be changed by JOD it's a good idea to set them `READONLY` because:

1. `READONLY` dictionaries can be accessed by any number of JOD tasks. `READWRITE` dictionaries can only be accessed by one task.
2. Keeping libraries `READONLY` prevents accidental `put`'s as you open and close dictionaries.

```
NB. make bptest READONLY
od 'bptest' [ 3 od ''
dpset 'READONLY'
```

Keep references updated. JOD stores word references: see `globs` on page 31. References enable many useful operations. References allow `getrx`, see page 30, to load words that call other words in new contexts.

```
NB. only put dictionary references need updating
0 globs&> }. revo ''
```

Document dictionary objects. Documentation is a long standing sore point for programmers. Most of them hate it. Some claim it's unnecessary and distracting. Many put in half-assed efforts. In my opinion this is "not even wrong!" Good documentation elevates code. In Knuth's [8] opinion it separates "literate programming" from the odious alternative. JOD provides a number of easy ways to document code: see `doc` on page 23, `put` on page 44 and `nw` on page 41.

Define your own JOD shortcuts. JOD words can be used within arbitrary J programs. If you don't find a JOD primitive that meets your needs do a little programming.

```
NB. describe a JOD group - (ijod) is JOD's interface locale
hg_ijod_=: [: hlpnl [: }. grp

NB. re-reference put dictionary show any errors
reref_ijod_=: 3 : '(n,.s) #~ -.;0{"1 s=.0 globs&>n=.}.revo''''' [ y'

NB. words referenced by group words that are not in the group
jodg_ijod_=: 'agroup'
nx_ijod_=: 3 : '(allrefs  }. gn) -. gn=. grp jodg'

NB. missing from (agroup)
nx ''
```

Customize JOD edit facilities. The main JOD edit words nw on page 41 and ed on page 26 can be customized by defining a DOCUMENTCOMMAND script.

```
NB. define document command script - {~N~} is word name placeholder
DOCUMENTCOMMAND_ijod_ =: 0 : 0
smoutput pr '{~N~}'
)

NB. edit a new word - opens edit window
nw 'bpword'
```

Define JOD project macros. When programming with JOD you typically open dictionaries, load system scripts and define nouns. This can be done in a project macro script: see rm on page 49.

```
NB. define a project macro - I use the prefix prj for such scripts
prjsunmoon=: 0 : 0

NB. standard j scripts
require 'debug task'

NB. local script nouns
jodg_ijod_=: 'sunmoon'
jods_ijod_=: 'sunmoontests'

NB. put/xref
DOCUMENTCOMMAND_ijod_ =: 'smoutput pr ''{~N~}'''
)

NB. store macro
4 put 'prjsunmoon';JSCRIPT_ajod_;prjsunmoon

NB. setup project
rm 'prjsunmoon'  [ od ;:'bpcopy bptest' [ 3 od ''
```

Maintain a make load scripts macro or test. To simplify the maintenance of JOD generated load scripts create a macro or test script that rebuilds load scripts when executed with rm on page 49 or rtt on page 49 .

```
NB.*make_load_scripts s-- generates load scripts.
NB.
NB. Can run as a tautology test see: rtt
```

```
cocurrent 'base'
require 'general/jod'
coclass 'AAAmake999' [ coerase <'AAAmake999'

>0{OPENDIC=: did 0
>0{od 'utils' [ 3 od ''

NB. utils scripts
>0{mls 'bstats'
>0{mls 'xmlutils'
>0{3 od ''

NB. jod tester
>0{od ;:'joddev jod utils'
>0{mls 'jodtester'
>0{3 od ''

NB. exif image utils
>0{od ;:'smugdev smug image utils'
>0{mls 'exif'
>0{3 od ''

cocurrent 'AAAmake999'
>0{od }. OPENDIC

cocurrent 'base'
coerase <'AAAmake999'
```

Edit your `jodprofile.ijs`. When JOD loads the profile script

```
~addons/general/jod/jodprofile.ijs
```

is run: see appendix F on page 68. Use this script to customize JOD. Note how you can execute project macros when JOD loads with sentences like:

```
NB. open required dictionaries and run project macro
rm 'prjsunmoon' [ od ;:'bpcopy bptest'
```

Use JOD documentation. JOD documentation is available as a PDF document `jod.pdf` than can be read with `jodhelp`: see page 35.

```
NB. display JOD help - requires general/joddocument addon
jodhelp 0
```

5 JOD Interface Words

This section describes all JOD user interface words in alphabetical order. Each word description consists of a short explanation followed by examples. Examples show J inputs; outputs are omitted. Word arguments are summarized with a form of Hungarian notation: see appendix J on page 76.

When JOD loads it creates a number of locale classes, (see appendix B on page 62), and defines an **ijod** user interface locale: **ijod** is placed on the **base** locale's path. JOD's **ijod** user interface consists of definitions like:

```
   get     NB. display get interface
get_1_ ::jodsf

   mls     NB. display mls interface
mls_8_ ::jodsf

   jodsf   NB. display error trap
0"_ ; 'JOD SYSTEM FAILURE: last J error -> '"_ , [: 13!:12 ''"_ [ ]
```

Each interface word calls a corresponding object instance word. The default interface is an *error trapping* interface.[8] If any JOD word fails `jodsf` will catch the error and return JOD's standard `(paRc;clMessage)` result.

5.1 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'
```

[8] Provisions for defining a *non-error-trapping* interface exist in JOD source code.

5.2 bget — backup get NIMP

5.3 bnl — backup name lists NIMP

5.4 compj — compress J code

compj compresses J code by removing comments, white space and shortening safe local identifiers to single characters.[9] 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 192 kilobytes when squeezed with compj it shrinks to about 72 kilobytes. compj does not compress words in JOD dictionaries it returns a compressed script result.

Warning: to safely use compj you must understand how to:

1. mark ambiguous names.
2. exclude words with local names that match quoted text.

If you do not properly mark ambiguous names and exclude words with local names that match quoted text compj will break your code!

Prior to compressing a word apply globs, see page 31, to expose any name problems.

compj compression can be limited to *white space removal* by marking explicit words with the comment tag *(-.)=:*.

```
donotsqueeze=: 4 : 0

NB. I have local names that occur in quoted strings
NB. Compressing me will end civilization as we know it.
NB. Use the comment tag (-.)=: anywhere to save the day.

NB. do not compress (-.)=:
home=. 2 * y
heart=. x - 1
if. 0=|home do.
  'home is where the heart is'
else.
  'heart health helps'
end.
)
```

[9] If more than one character is required to rename all local identifiers compj uses a letter prefixed high base numbering scheme.

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. Comment scope tags override J scope.

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

The following examples show 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. be used out of nowhere
one * two * three
we + are + hidden + locals
)

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 instance
```

```
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 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 jodijs.zip from *The JOD Page* [6]. JOD source is distributed as JOD dictionary dump scripts and as fully commented class files in jodijs.zip.

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'
```

Dyad: *iaOption compj clName ∨ blclNames*

```
NB. remove comments preserving leading whitespace and
NB. original identifiers - useful for reading code
NB. without distracting comments
>1{ 1 compj 'justthecode'
```

5.5 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 *nonput 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.[10] 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.

[10] In database terms del can violate *referential integrity*. Early versions of JOD maintained referential integrity but this proved cumbersome and was dropped.

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 word references
11 del ;:'remove our references'
```

5.6 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 not 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'
```

5.7 did — dictionary identification

did identifies 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
```

5.8 disp — display dictionary objects

disp displays dictionary objects. disp returns a character list when successful and the standard boxed (paRc;clMessage) 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 group header
2 1 disp 'group'

NB. display group documentation
2 9 disp 'group'

NB. display all short group explanations
2 8 disp }. grp ''

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

NB. display suite header
3 1 disp 'suite'

NB. display suite documentation
3 9 disp 'suite'

NB. display one macro
4 disp 'macro'

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

NB. display put dictionary documentation
5 disp ''
```

5.9 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 ∨ clPstr`*

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

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

Dyad: *`iaObject dnl zl ∨ clPstr`*
`(iaObject,iaOption) dnl zl ∨ 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 2 3 dnl 'str'

NB. nouns ending with str
0 3 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'
```

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

```
NB. words containing str (result is a list of lists)
0 _2 3 dnl 'str'

NB. group names beginning with so
2 _1 dnl 'so'

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

5.10 doc — format comments

doc formats the leading comment block of explicit words, tests, group/suite headers and macros. The comment block must follow J scriptdoc [1] 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. Install the jodsource addon to get 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
NB.     mess is cleaned up. See below.
NB. monad: docexample uuHungarian
NB.
NB.   text below monad: and dyad: is left unformatted
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: is left unformatted
NB.   this region is used to display example calls
j code from now on
)
```

Monad: *doc clName*

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

Dyad: *iaObject doc clName*
(iaObject,iaOption) doc clName

```
NB. format leading test script comments
1 doc 'tidytest'

NB. format J script macros - only J scripts formatted
4 doc 'macroname'

NB. group and suite headers
2 1 doc 'thisgroup'
3 1 doc 'thissuite'
```

doc also formats the long document text of all dictionary objects. Long document text differs from J script text in that it is not prefixed with leading NB.'s.

```
NB. format long documents
0 9 doc 'word'
1 9 doc 'test'
2 9 doc 'group'
3 9 doc 'suite'
4 9 doc 'macro'
```

5.11 dpset — set and change parameters

dpset modifies dictionary parameters. JOD uses a variety of values that control putting, getting and generating objects. Parameters are stored in individual dictionaries and the master file. Master file parameters are initially set from the jodparms.ijs file, see appendix E, on page 66, and cannot be reset without editing jodparms.ijs and recreating the master file. Individual dictionary parameters can be changed at any time with dpset. 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. dpset 'RESETME' and dpset 'RESETALL' clear 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 task never use RESETALL unless you are absolutely sure you will not reset other tasks!

```
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. use ascii85 in dump scripts
dpset 'ASCII85';1
```

5.12 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'
```

5.13 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 (1!:1) <'c:\temp\text.txt'
```

5.14 gdeps — group dependents

gdeps returns lists of global names in the dependent section of group and suite headers: see page 52.

Monad: *gdeps clGroup*

```
NB. globals in the dependent section of group jod
gdeps 'jod'

NB. all dependent section globals in all groups
gdeps&> }. grp ''
```

Dyad: *iaOption gdeps clName*

```
NB. globals in the dependent section of suite testenv
3 gdeps 'testenv'
```

5.15 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 page 44. A basic JOD rule is that if you can put it you can get it.

Monad: *get clName ∨ blclNames*

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

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

Dyad: *ilOptions get clName ∨ blclNames ∨ uuIgnore*
clLocale get clName ∨ blclNames

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

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

For words a character left 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'
```

Information about stored 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'
```

get fetches 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 explanation 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'

NB. get dictionary documentation - ignores (y) argument
5 get ''
```

5.16 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 ∨ 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'
```

5.17 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:

```
jread_jfiles_     NB. direct locale reference
did__jd2          NB. indirect locale (object) reference
did_3_            NB. direct numbered local reference
boo__hoo__too     NB. two levels of indirection

NB.  suffix nouns: jd2 hoo too
```

are treated like primitives. This makes it possible to define clean locale/object interfaces. In the case of indirect locale references the suffix noun(s) 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_   NB. z interface for jread
```

Words that use jread can call it without 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 reference 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'
```

5.18 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*

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

NB. list words in group
grp 'group'

NB. create empty group
grp 'soempty';''

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*

```
NB. list all test 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'
```

5.19 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'
```

5.20 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'
```

5.21 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.[11]

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 ''
```

5.22 jodhelp — display help

jodhelp displays JOD documentation. The monad starts the configured J PDF reader and displays jod.pdf.

jod.pdf is omitted from the core JOD addon JAL package to reduce download size. Installing the addon general/joddocument saves jod.pdf in the directory searched by jodhelp.

```
NB. JOD addon PDF document directory
~addons/general/joddocument/pdfdoc
```

Monad: *jodhelp uuIgnore*

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

```
NB. display help - requires general/joddocument addon
jodhelp 0

NB. argument is ignored
jodhelp 'I don''t hear you!'
```

5.23 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 J macro scripts as postprocessors.[12] 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'
```

5.24 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.

[12] A macro must be coded as a J script, (code 21), to be used as a postprocessor.

Monad: *locgrp clName*

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

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

5.25 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 subsection 6.1 on page 51.

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 groups and suites 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 Windows file
make 'c:/dump/on/me.ijs'

NB. linux and mac paths must begin with /
make '/home/john/temp/joddumps/metoo.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: see page 39.

```
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'
```

5.26 mls — make load script

mls generates J load scripts. The generated script is added to the current user's start up script

```
~config/startup.ijs
```

and inserted in the session's Public_j_ table.[13]

An mls load script is independent of JOD and can be used like any other J load script, for example:

```
load 'mlsmademe'
```

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 subsection 6.1 Generated Script Structure, on page 51.

[13] On J 6.0x systems the public script noun is named PUBLIC_j_.

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 startup.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 startup.ijs
0 mls 'bstats'

NB. monad
1 mls 'bstats'

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

5.27 mnl — master name lists NIMP

5.28 newd — create a new dictionary

newd creates a new dictionary. Dictionary creation generates a set of files in a standard dictionary directory structure: see Figure 3 on page 53. 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/'
newd 'deep';'d:/we/can/root/dictionaries/down/deep'

NB. on linux create dictionaries in your $HOME directory
newd 'homeboy';'/home/john/where/the/dictionaries/are'

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

5.29 notgrp — not grouped

notgrp list words or tests from (y) that are not in groups or suites. Useful for finding loose ends and dead code.

Monad: *notgrp clName ∨ blclNames*

```
NB. recent ungrouped words
notgrp }. revo ''

NB. all ungrouped words
notgrp }. dnl ''
```

Dyad: *iaOption notgrp clName ∨ blclNames*

```
NB. ungrouped words (monad)
2 notgrp }. dnl ''

NB. tests that are not in suites
3 notgrp }. 1 dnl ''
```

5.30 nw — edit a new word

nw edits a new word in an edit window using JOD format conventions. nw assumes the new word is *explicit* but you can edit the default text to define *tacit* words. nw will append the character list DOCUMENTCOMMAND to the text placed in an edit window. This allows the user to define an arbitrary script that is run when a word is defined.[14]

Monad: *nw clName*

```
NB. open an edit window with an explicit newword
nw 'newword'

NB. define a script that is run when the J editor
NB. window is run. {~N~} placeholders are replaced
NB. with the name of the new word

DOCUMENTCOMMAND_ijod_=: 0 : 0
smoutput pr '{~N~}'
)

NB. edit a word with DOCUMENTCOMMAND
nw 'placeholdername'
```

Dyad: *iaNameclass nw clName*

```
NB. edit an explict adverb
1 nw 'adverb'

NB. edit an explict conjunction
2 nw 'conjunction'

NB. edit an explicit text noun
0 nw 'text'

NB. edit an explicit word (monad)
3 nw 'word'

NB. edit dyadic word
4 nw 'dyad'
```

[14] Key combinations like CTRL-W that run editor scripts depend on the J platform.

5.31 nt — edit a new test

nt edits a new test script in an edit window. nt looks for the test script teststub on the path and inserts teststub text in the edit window. teststub allows users to define dictionary specific custom script formats.

nt searches teststub text and replaces the markers {˜T˜}, {˜A˜}, {˜D˜} and {˜SD˜} with optional *title*, *author*, *date*, (yyyymondd format), and *short date*, (yymondd format). The dyad makes additional delimited string replacements after processing markers.

Monad: *nt clTestName*

```
NB. open an edit window
nt 'newtest'

testheader=: 0 : 0
NB.*{˜T˜} t-- custom test script header
NB.
NB. This is my custom test script header.
NB. The {˜T˜} strings are replaced with test
NB. name and creation time {˜D˜}
NB.
NB. author: {˜A˜}
NB. created: {˜D˜}
)

NB. save custom header in put dictionary
1 put 'teststub';testheader

NB. CLASSAUTHOR replaces {˜A˜}
CLASSAUTHOR_ijod_=: 'Mansong Hydrogen'

NB. edit a new test using the custom header
NB. where title {˜T˜} is set from the (y) argument
nt 'customtest'
```

Dyad: *clReplacements nt clTestName*

```
NB. apply string replacements, delimited by first
NB. character, to teststub text then open edit window
'#I#can#CHANGE#you' nt 'newtest'
```

5.32 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 d(i) read/write
od 'd1';'d2';'d3'
```

Dyad: *iaOption od zl ∨ clDictionary ∨ 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 d(i) 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 ''
```

5.33 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 tasks point to the same master file.[15]

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'
```

5.34 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: see page 28.

Monad: *put clName ∨ blclNames*

[15] There is one important exception to the single jmaster.ijf rule. READONLY dictionaries can be safely registered in different master files. This allows easy sharing of READONLY library dictionaries on network drives with the standard JOD/JAL setup.

```
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*
iaObject put clDocument ∨ 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 ...')
```

A group or suite header script is an arbitrary J script that preceeds the code generated by `make` on page 37.

```
NB. put group header scripts from name,value table
2 put (;:'g0 g1'),.('text ...';'text ...')
```

```
NB. group header scripts can be put
NB. with 2 1 as well - maintains put/get symmetry
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 ...';'...')
```

The text types `LaTeX, HTML, XML, ASCII, BYTE, MARKDOWN, UTF8` are stored as J character lists.

```
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. list of arbitrary bytes
4 put (;:'m0 m1'),.(26;26),.('bytes ...';'...')

NB. UTF-8 unicode text
4 put (;:'m0 m1'),.(28;28),.('utf8 text ...';'...')

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

```
4 9 put (;:'m0 m1'),.('text ...';'text ...')

NB. update dictionary documentation text
5 put 'go ahead document this dictionary'

NB. dictionary documentation is controlled
NB. by DOCUMENTDICT with default 1
dpset 'DOCUMENTDICT';0
5 put 'this will not be stored'
```

5.35 regd — register dictionaries

regd registers and unregisters dictionaries in the master file.

A dictionary is a set of files in a standard directory structure: see page 53. 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 linux dictionary with optional documentation
regd 'name';'/home/john/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
```

5.36 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'
```

5.37 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'
```

5.38 rm — run macros

A JOD macro is an arbitrary J script. rm fetches J macro scripts and runs them. rm will only run J, code 21, macros; other types return errors.

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'
```

5.39 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'
```

5.40 rxs — regular expression search NIMP

5.41 uses — return word uses

uses lists words used by other words. The lists are derived from the cross references stored 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 is 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'
```

6 JOD Scripts

6.1 Generated Script Structure

To use dictionary words it is necessary to generate scripts. JOD scripts come in three flavors:

1. Arbitrary J scripts
2. Header and list scripts
3. Dump scripts

JOD test, macro and group/suite headers are arbitrary J scripts. There are no restrictions on these scripts. Group and suite scripts generated by `make`, `mls` and `lg`, (see pages 37, 38, 36), are header and list scripts. `make` produces dump scripts.

JOD script structure mirrors what you typically do in a J application script. With most J application scripts you:

1. Setup the application's runtime environment.
2. Load the classes, words and data that comprise the application.
3. Start the application.

This pattern of *setup*, *load* and *start* is seen over and over in J scripts: see Table 2 on page 51.

Generated Script Structure			
Section	**Type**	**Description**	**Example**

Setup	*Active*	Define group and Suite headers. Headers may contain one dependent section: see page 52.	`NB. define a group header`
			`2 1 put 'groupname';' ... script text ... '`
			
			`NB. define a suite header`
			`3 1 put 'suitename';' ... suite text ... '`

Load	*Passive*	Load lists of words or tests. *Only word lists are passive.* Tests are typically active scripts.	`NB. form group from stored words`
			`grp 'groupname' ; ;:'words in group'`
			
			`NB. form suite from stored tests`
			`3 grp 'suitename' ; ;:'stored tests'`

Start	*Active*	Associate a postprocessor macro with a group or suite. Postprocessors are prefixed with `POST_`	`NB. group postprocessor`
			`4 put 'POST_groupname';21;' ... script ... '`
			
			`NB. test suite postprocessor`
			`4 put 'POST_suitename';21;' ... script ... '`

Table 2: JOD generated script structure

6.2 Dependent Section

A dependent section is a delimited subsection of a group or suite header, (see grp on page 32), that is used to define related words and runtime globals. *Global words defined in a dependent section are removed from group lists when groups are generated with make, mls and lg: see pages 37, 38 and 36.* This insures that the values assigned in the dependent section are maintained when the group script loads.

A dependent section is delimited with NB.*dependents and NB.*enddependents and only one dependent section per group header is allowed. The following is the dependent section in the jod class group header. Globals in a dependent section are returned by gdeps, see page 27.

```
NB.*dependents x-- words defined in this section have related definitions
NB. host specific z locale nouns set during J profile loading
NB. (*)=: IFWIN UNAME
LF=:10{a.
CR=:13{a.
TAB=:9{a.
CRLF=:CR,LF
NB. option codes - to add more add a new object code
NB. and modify the following definition of MACROTYPE
JSCRIPT=:21
LATEX=:22
HTML=:23
XML=:24
TEXT=:25
BYTE=:26
UTF8=:28
NB. macro text types, depends on: JSCRIPT,LATEX,HTML,XML,TEXT,BYTE,UTF8
MACROTYPE=:JSCRIPT,LATEX,HTML,XML,TEXT,BYTE,UTF8
NB. object codes
WORD=:0
TEST=:1
GROUP=:2
SUITE=:3
MACRO=:4
NB. object name class, depends on: WORD,TEST,GROUP,SUITE,MACRO
OBJECTNC=:WORD,TEST,GROUP,SUITE,MACRO
NB. bad object code, depends on: OBJECTNC
badobj=:[: -. [: *./ [: , ] e. OBJECTNC"_
NB. path delimiter character & path punctuation characters
PATHDEL=: IFWIN { '/\'
PATHCHRS=:' :.-',PATHDEL
NB. default master profile user locations
JMASTER=:jodsystempath 'jmaster'
JODPROF=:jodsystempath 'jodprofile.ijs'
JODUSER=:jodsystempath 'joduserconfig.ijs'
NB.*enddependents
```

7 JOD Directory and File Layouts

JOD stores J objects in binary `jfiles`. When newd creates a dictionary it registers the location of the dictionary in `jmaster.ijf`, see Table 3 on page 54, and creates a set of standard directories: see Figure 3 on page 53. This section describes the internal structure of JOD's binary `jfiles`.

JOD Directory Structure created by `newd 'name';'.../'`

`.../name` The dictionary root directory holds binary `jfiles`, see section 7 on page 53.

```
jwords.ijf, jtests.ijf
jgroups.ijf, jsuites.ijf
jmacros.ijf
juses.ijf
```

- **`name/alien`** Unused — reserved for files related to this dictionary.
- **`name/backup`** Binary backups created by `packd` (44).
- **`name/document`** Unused — reserved for documents related to this dictionary.
- **`name/dump`** J dump script files generated by `make` (37).
- **`name/script`** J script files generated by `make`, `mls` and `lg` (37, 38, 36).
- **`name/suite`** J test scripts generated by `make` (37).

Figure 3: `newd` generates this directory structure when a new JOD dictionary is created. The locations of JOD directories are stored in directory objects when dictionaries are opened.

7.1 Master File — `jmaster.ijf`

`jmaster.ijf` 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 3 on page 54.

7.2 Words File — `jwords.ijf`

`jwords.ijf` is a binary component `jfile`. `jwords.ijf` contains word definitions and metadata. The component layout of `jwords.ijf` is given in Table 4 on page 55.

7.3 Tests File — `jtests.ijf`

`jtests.ijf` is a binary component `jfile`. `jtests.ijf` contains test definitions and metadata. The component layout of `jtests.ijf` is given in Table 5 on page 56.

jmaster.ijf		
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,X)	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 (pp. 43) the status is set and stays on until closed by od. This blocks all other dictionary tasks from updating 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}	Xl	Dictionary log. The dictionary log is a simple, (append only), list of all the extended dictionary numbers (DIDNUMs) 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 3: jmaster.ijf file component layout

jwords.ijf		
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	cl	Dictionary documentation newd, regd. (pp. 39,47)
c_3	bluu	Dictionary parameters. 0 { - cl ; dictionary name 1 { - Xa ; dictionary number DIDNUM (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 (pp. 54). 0 { cl ; parameter 1 { uu ; value
		Main inverted items, $c_4 \rightarrow c_{11}$ 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.
$c_{12} \rightarrow c_{38}$		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 ∨ uu ; word value: nouns binary, all others 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 4: jwords.ijf file component layout

jtests.ijf		
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 5: jtests.ijf file component layout

7.4 Groups File — jgroups.ijf

jgroups.ijf is a binary component jfile. jgroups.ijf contains group definitions and group metadata. The component layout of jgroups.ijf is given in Table 6 on page 57.

jgroups.ijf		
Component	**Hungarian**	**Description**
c_0	blnl	Group 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	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).
$c_9 \rightarrow c_{10}$		Reserved.
c_{11}	blcl	Short group explanations.
$c_{12} \rightarrow c_{38}$		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 6: jgroups.ijf file component layout

7.5 Suites File — jsuites.ijf

jsuites.ijf 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 7 on page 58.

jsuites.ijf		
Component	**Hungarian**	**Description**
c_0	blnl	Suite 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	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).
$c_9 \rightarrow c_{10}$		Reserved.
c_{11}	blcl	Short suite explanations.
$c_{12} \rightarrow c_{38}$		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 7: jsuites.ijf file component layout

7.6 Macros File — jmacros.ijf

jmacros.ijf 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 8 on page 59.

jmacros.ijf		
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 8: jmacros.ijf file component layout

7.7 Uses File — juses.ijf

juses.ijf is a binary component jfile. juses.ijf contains word references: see globs subsection 5.17, on page 31.

juses.ijf		
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.
$c_1 \rightarrow c_4$		Reserved.
		Uses (reference) directory layout differs from jwords.ijf but occupies the same component range for packd (pp. 44). Only non-empty reference lists are stored.
c_5	blcl	Word uses words (index).
c_6	il	Component list.
$c_7 \rightarrow c_{18}$		Reserved.
c_{19}	Xl	Put reference path. List of extended dictionary numbers DIDNUMs.
$c_{20} \rightarrow c_{38}$		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 9: juses.ijf file component layout

A JOD Distribution

JOD is distributed as a *J addon*. You can install JOD using JAL the *J package manager* [10].

The JOD distribution is broken into three JAL packages:

1. jod [4]: This is the only package that must be installed to run JOD. It contains JOD system code and supporting files.

2. jodsource [5]: This addon consists of three JOD dictionary dumps and a setup script. 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 test script dictionary

   (c) utils.ijs — common utilities dictionary

   (d) jodsourcesetup.ijs — J script that creates and loads the three JOD development dictionaries.

3. joddocument [7]: this package contains JOD PDF documents. Installing this package places these documents on local drives for jodhelp, see page 35.

B JOD Classes

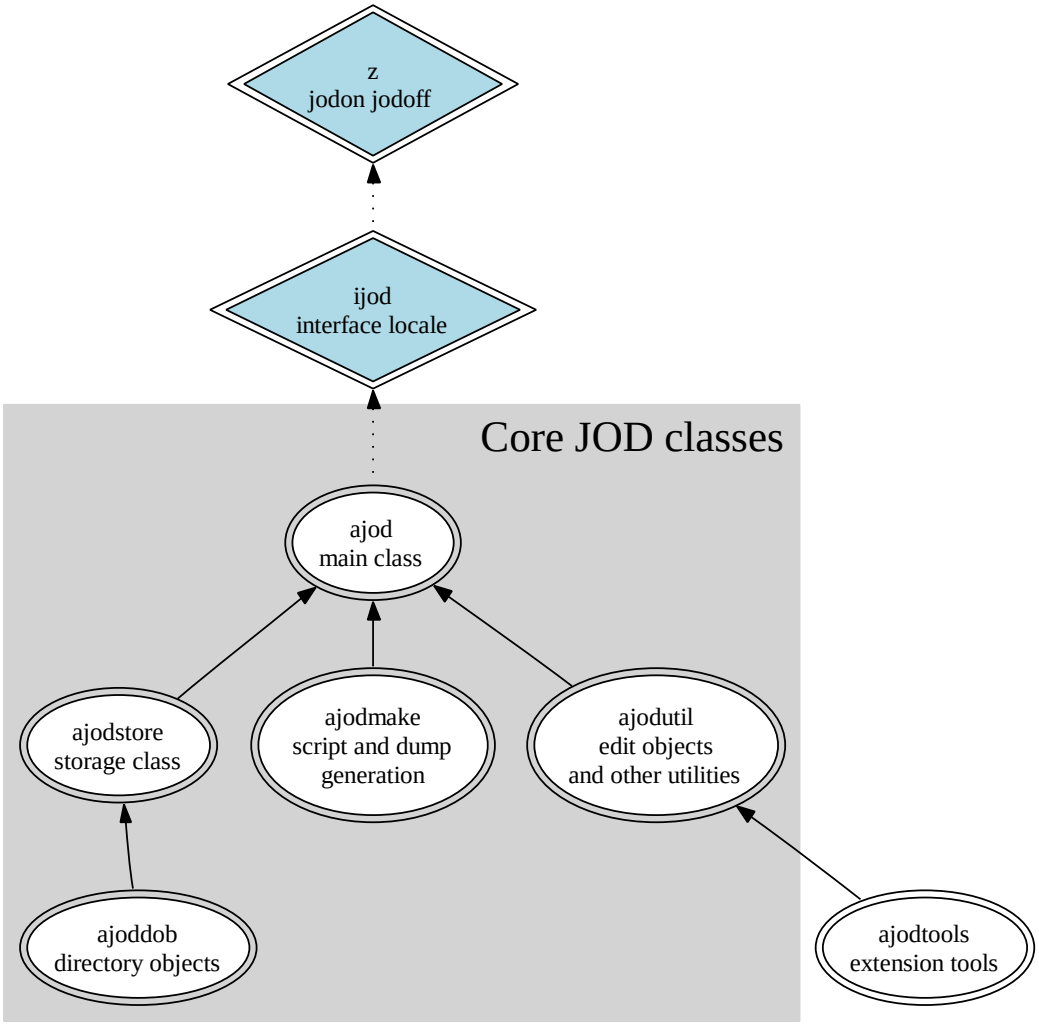


Figure 4: This diagram shows how JOD classes are related. JOD classes are loaded into J addon **a** locales. The arrows indicate how J names are resolved. *Diagram Generated by* `Graphviz` [9].

C Reference Path

JOD groups and suites, (see grp on page 32), are defined with respect to a particular path. This path is called the *reference path*. The reference path is stored when the first put dictionary group or suite is defined. Group and suite generation with make, mls and lg, (see pages 37, 38, 36), check the current path against the reference path. If the paths do not match an error is returned.

Reference paths display current dictionary names but the path is stored as a unique list of extended dictionary identification numbers: DIDNUMs. On Windows and Linux systems the DIDNUM is based on GUIDs. DIDNUMs insure reference paths are unique.

A reference path can only be reset by clearing the put dictionary path, opening desired dictionaries and recreating a group or suite: see dpset on page 25.

```
    NB. open first five dictionaries
    od 5 {. }. od ''
+-+-------------------------+------+---+-----+--------+---+
|1|opened (rw/rw/ro/rw/rw) ->|budget|cbh|flick|flickdev|gps|
+-+-------------------------+------+---+-----+--------+---+

    NB. display dictionary information - reference paths in last column
    did ~ 0
+-+-----------------------------------------------------------------------+
|1|+--------+--+-----+-----+-------+-------+------+---------------------+|
| ||        |--|Words|Tests|Groups*|Suites*|Macros|*Path                ||
| |+--------+--+-----+-----+-------+-------+------+---------------------+|
| ||budget  |rw|14   |0    |2      |0      |0     |/budget              ||
| |+--------+--+-----+-----+-------+-------+------+---------------------+|
| ||cbh     |rw|145  |0    |6      |0      |6     |/cbh/utils           ||
| |+--------+--+-----+-----+-------+-------+------+---------------------+|
| ||flick   |ro|296  |3    |9      |0      |9     |/flick/utils         ||
| |+--------+--+-----+-----+-------+-------+------+---------------------+|
| ||flickdev|rw|96   |2    |2      |0      |2     |/flickdev/flick/utils||
| |+--------+--+-----+-----+-------+-------+------+---------------------+|
| ||gps     |rw|11   |0    |0      |0      |0     |/gps/utils           ||
| |+--------+--+-----+-----+-------+-------+------+---------------------+|
+-+-----------------------------------------------------------------------+
```

D JOD Argument Codes

The left, and some right, arguments of JOD 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 codes: see tables 10, 11 and 12 on pages 64, 64 and 65.

Object Codes			
Noun	**Code**	**Use**	**Example**
WORD	0	word code	`0 dnl ''  NB. list all words on path`
TEST	1	test case code	`1 put 'test';'test code..'  NB. store test`
GROUP	2	group code	`2 put 'name';'group header ...'  NB. store group header`
SUITE	3	suite code	`3 grp 'suite'  NB. get suite members, list of test names`
MACRO	4	macro code	`4 disp 'test'  NB. display macro`
DICTIONARY	5	dictionary code	`5 get ''  NB. get dictionary documentation`

Table 10: JOD Object Codes

Qualifier Codes			
Noun	**Code**	**Use**	**Example**
DEFAULT	7	default action	`0 7 get 'this'  NB. default behaviour`
EXPLAIN	8	short explanation text	`0 8 put 'name';'explain name'`
DOCUMENT	9	documentation text	`2 9 put 'group';'very long group document ...'`
NVTABLE	10	name value table	`0 10 get }. dnl ''  NB. return all words in table`
REFERENCE	11	reference code	`11 del 'earthdist'  NB. delete word references`
JSCRIPT	21	J script code	`4 1 21 dnl 'POST_'  NB. list postprocessors`
LATEX	22	LaTeX 2ε text code	`4 get }. 4 3 22 dnl 'TEX'  NB. get LaTeX macros`
HTML	23	HTML text code	`4 put 'htmltxt';23;'<a>hello world</a>'  NB. store html`
XML	24	XML text code	`4 put 'xmltxt';24;'<p>baby step xml</p>'  NB. store xml`
TEXT	25	ASCII text code	`4 3 25 dnl 'EPS'  NB. texts ending with EPS`
BYTE	26	BYTE characters	`4 put 'BYTEME';a.`
MARKDOWN	27	MARKDOWN text code	`5 put 'Main **dictionary** document'`
UTF8	28	Unicode UTF8 text	`4 put 'UTF8TEXT';UTF8_ajod_;(8 u: 4 u: 56788 4578,65+i.5)`

Table 11: JOD Qualifier Codes

The meaning of negative option and qualifier codes depends on the word. For `dnl` a negative option requests a *path order list.*

Negative Codes		
Code	**Use**	**Example**
`_1`	path order list	`0 _1 dnl ''  NB. path order list of words`
`_2`	path order list	`1 _2 dnl 'boo'  NB. path order list of test names containing boo`

Table 12: JOD Negative Codes

E jodparms.ijs

`jodparms.ijs` is read when the master file `jmaster.ijf` is created and is used to set dictionary parameters.

Dictionary parameters are distributed to dictionary files and runtime objects. New parameters can be added by editing `jodparms.ijs` and recreating the master file. The last few lines of the following example show how to add `COPYRIGHT` and `MYPARAMETER`.

When a parameter is added its value will appear in the directory objects of all dictionaries but will only be `dpset`'able in new dictionaries.

To change default master dictionary parameters:

1. Exit J
2. Delete the files

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

3. Edit

   ```
   ~addons/general/jod/jodparms.ijs
   ```

4. Restart J and reload JOD with

   ```
   load 'general/jod'
   ```

```
NB.*jodparms s-- default dictionary parameters.
NB.
NB. This file is used to set  the  default  dictionary parameters
NB. table  in the master file. When  a  new dictionary is created
NB. the  parameters  in the  master file are used to specify  the
NB. parameters for a particular dictionary. The  verb (dpset) can
NB. be   used  to   modify  parameter  settings   in   individual
NB. dictionaries.  Master  file parameters can only be changed by
NB. editing this file and recreating the master file.
NB.
NB. WARNING: all the  parameters currently listed are required by
NB. the JOD system. If you remove any of them JOD will crash. You
NB. can  safely add additional parameters but  you  cannot safely
NB. remove current parameters.

MASTERPARMS=: 0 : 0

NB. The format of this parameter file is:
NB.     jname ; (type) description ; value
NB.
NB.     jname is a valid J name
```

```
NB.     (type) description documents the parameter - type is required
NB.        only (+integer) is currently executed other types will
NB.        be passed as character lists (see dptable).
NB.     value is an executable J expression that produces a value

PUTFACTOR  ; (+integer) words stored in one loop pass  (10<y<2048)      ; 100
GETFACTOR  ; (+integer) words retrieved in one loop pass (10<y<2048)    ; 250
COPYFACTOR ; (+integer) components copied in one loop pass  (1<y<240)   ; 100
DUMPFACTOR ; (+integer) objects dumped in one loop pass (1<y<240)       ; 50
DOCUMENTDICT ; (+integer) when 1 dictionary document is put (0 or 1)    ; 1
DOCUMENTWIDTH ; (+integer) width of justified document text  (20<y<255) ; 61
ASCII85    ; (+integer) when 1 ascii85 used in dumps - default 0        ; 0

NB. Any added parameters are stored in the master file when
NB. created and distributed to JOD directory objects.

NB. WARNING: when defining J expressions be careful about the ; character
NB. the JOD code (dptable) that parses this string is rudimentary.
NB. VISITOR ; (character) executed by directory object visitor ; dropdir 0

NB. COPYRIGHT ; (character) ; All rights reserved
NB. MYPARAMETER ; (+integer) the answer ; 42
)
```

F jodprofile.ijs

`jodprofile.ijs` is an optional user profile script; it runs after JOD loads and can be used to customize your working environment. The following is an example profile script.

```
NB.*jodprofile s-- JOD dictionary profile.
NB.
NB. An example JOD profile script. Save this script in
NB.
NB. ~addons/general/jod
NB.
NB. with the name jodprofile.ijs
NB.
NB. This script  is  executed  AFTER all dictionary  objects have
NB. been created. It  can  be used  to  set up  your default  JOD
NB. working environment.
NB.
NB. WARNING: Do not dpset 'RESETME' if more than one  JOD task is
NB. active. If only one task is active RESETME's prevent annoying
NB. already open  messages that frequently result from forgetting
NB. to close dictionaries upon exiting J.

NB. set white space preservation on
9!:41 [ 1

NB. do not reset if you are running more than one JOD instance
dpset 'RESETME'

NB. project shortcuts - use explicit
NB. defintions so it's easy to reset the group/suite
ag_ijod_=: 3 : 'jodg addgrp y'
dg_ijod_=: 3 : 'jodg delgrp y'

NB. referenced group words not in group
nx_ijod_=: 3 : '(allrefs  }. gn) -. gn=. grp jodg'

NB. short help for group words
hg_ijod_=: [: hlpnl [: }. grp

NB. regenerate put dictionary word cross references
reref_ijod_=: 3 : '(n,.s) #~ -.;0{"1 s=.0 globs&>n=.}.revo'''' [ y'

NB. open working dictionaries and run project macros
NB. set up current project (1 suppress IO, 0 or elided display)
NB. 1 rm 'prjThumbutilsSetup' [ smoutput od ;: 'imagedev image utils'
NB. 1 rm 'prjjod' [ smoutput od ;:'joddev jod utils'
```

G joduserconfigbak.ijs

joduserconfigbak.ijs is an optional configuration script. joduserconfigbak.ijs is in the directory.

```
~addons/general/jod/jodbak
```

joduserconfigbak.ijs can be used to redefine class words before any JOD objects are created.

```
NB.*joduserconfigbak s-- example JOD user configuration script.
NB.
NB. This  script is  executed BEFORE JOD objects  are created. It
NB. can be used to redefine and customize various class nouns. To
NB. make  this script  active  rename it to joduserconfig.ijs and
NB. copy it, with your edits, to the main jod directory:
NB.
NB. ~addons/general/jod
NB.
NB. The nouns listed below are good candidates for redefinition.

NB. smoutput 'joduserconfig.ijs executing ...'

NB. PDF reader in jodutil class
NB. Reset for new versions or other PDF readers
NB. PDFREADER_ajodutil_=:'C:\Program Files\Adobe\Reader 8.0\Reader\acrord32.exe'

NB. alternative Ghostscript compatible reader
PDFREADER_ajodutil_=:'C:\Program Files\Ghostgum\gsview\gsview32.exe'

NB. Preferred web browser in jodutil class
WWW0_ajodutil_=:'c:\Program Files\Mozilla Firefox\firefox.exe'

NB. Secondary web browser in jodutil class
WWW1_ajodutil_=:'C:\Program Files\Internet Explorer\IEXPLORE.EXE'

NB. Note: PDF reader and web browser values set with J's
NB. configuration utility take precedence over these nouns.
```

H JOD `startup.ijs` entries

`startup.ijs` is J's optional user startup script. `startup.ijs` is in the directory.

```
~config
```

JOD uses `startup.ijs` to store load scripts generated by `mls`: see page 38.

```
NB. WARNING: JOD managed section do not edit!
NB.<JOD_Load_Scripts>
buildpublic_j_ 0 : 0
bstats  c:/jod/jutils/script/bstats
xmlutils  c:/jod/utils/script/xmlutils
analystgraphs  c:/jod/franklin/script/analystgraphs
exif  c:/jod/smugdev/script/exif
jodtester  c:/jod/joddev/script/jodtester
waypoints  c:/jod/gps/script/waypoints
Weeks  c:/jod/docs/script/Weeks
MweccTeXPreprocess  c:/jod/docs/script/MweccTeXPreprocess
DudTeXPreprocess  c:/jod/docs/script/DudTeXPreprocess
BiblioHelper  c:/jod/docs/script/BiblioHelper
RecodeSchedZ  c:/jod/mwecc/script/RecodeSchedZ
)
NB.</JOD_Load_Scripts>
```

I JOD and Version Control Systems

Despite JOD's backup and restore facilities, see bnl, bget, packd and restd on pages 16, 16, 44 and 48, JOD is not a source code version control system like Git [12] or Subversion [14]. JOD's primary purpose is efficiently refactoring, shuffling and recombining J words not tracking their detailed histories. *Traditional version control systems focus on the history of source code* and provide detailed merge, security and multiuser network facilities that JOD lacks. However, since JOD generates standard J source code scripts it's easy to use JOD with version control systems. The main difficulty is choosing a suitable level of detail: *dictionary*, *script* or *word*. The following shows how Git can be used for each of these levels. Git has a number of graphical GUI interfaces these examples use bash shell commands.

1. **Dictionary:** make, see page 37, can dump entire dictionaries as a single J script. Dump scripts contain all[16] dictionary word definitions, test scripts, groups, suites and macros. Storing dump scripts in version control systems is an effective and simple way of tracking dictionary changes. To create dump scripts I run the macro dumpput. dumpput is stored in the utils dictionary; it dumps the put dictionary and copies the generated script to a common local directory. The common local directory hosts a Git repository that has a GitHub remote repository set. A remote GitHub repository is good way to move dictionaries between machines and safely share them with others. In the following example local changes are committed and then pushed to a remote repository.[17]

```
   NB. Step 1: J session commands - open dictionaries
   od ;:'docs utils' [ 3 od ''
+-+-----------------+----+-----+
|1|opened (rw/ro) ->|docs|utils|
+-+-----------------+----+-----+

   1 rm 'dumpput'  NB. run dump macro - (utils) must be on path
+-+--------------------------+------------------------+
|1|object(s) on path dumped ->|c:/jod/docs/dump/docs.ijs|
+-+--------------------------+------------------------+
+------------------------+
|c:/jod/joddumps/docs.ijs|
+------------------------+
```

```
$ echo Step 2: Bash shell commands > /dev/null

bakerjd99@NINJA /c/jod/joddumps (master)
$ pwd
/c/jod/joddumps
```

[16] Word references are not present in dump scripts. They can be easily regenerated with globs, see page 31.

[17] A collection of JOD dictionary dump scripts is available at: https://github.com/bakerjd99/joddumps.

```
bakerjd99@NINJA /c/jod/joddumps (master)
$ git status -s
M  docs.ijs
 M joddev.ijs
 M utils.ijs

bakerjd99@NINJA /c/jod/joddumps (master)
$ git commit -m 'recent changes to docs.ijs dictionary'
[master 1577d1a] recent changes to docs.ijs dictionary
 1 files changed, 46 insertions(+), 4 deletions(-)

bakerjd99@NINJA /c/jod/joddumps (master)
$ git remote
joddumps
origin

bakerjd99@NINJA /c/jod/joddumps (master)
$ git push joddumps master
```

2. **Script:** Word and test scripts generated by JOD are stored in a dictionary's `script` and `suite` subdirectories, see Figure 3 on page 53. In the following a `Git` repository has been created in the `script` subdirectory and the contents of the `exif` group have been edited and regenerated.

```
   NB. Step 1: J session commands - open dictionaries
   od ;:'smugdev smug image utils' [ 3 od ''
+-+----------------------+-------+----+-----+-----+
|1|opened (rw/ro/ro/ro) ->|smugdev|smug|image|utils|
+-+----------------------+-------+----+-----+-----+

   NB. edit (exif) content and save changes ...

   NB. regenerate (exif) script
   mls 'exif'
+-+--------------------+------------------------------+
|1|load script saved ->|c:/jod/smugdev/script/exif.ijs|
+-+--------------------+------------------------------+
```

```
$ echo Step 2: Bash shell commands > /dev/null

bakerjd99@NINJA /c/jod/smugdev/script (master)
$ pwd
/c/jod/smugdev/script

bakerjd99@NINJA /c/jod/smugdev/script (master)
$ git status -s
 M exif.ijs
```

```
bakerjd99@NINJA /c/jod/smugdev/script (master)
$ git add exif.ijs

bakerjd99@NINJA /c/jod/smugdev/script (master)
$ git commit -m '(masspixels) added to description of (exif) interface'
[master e93ffe5] (masspixels) added to description of (exif) interface
 1 files changed, 13 insertions(+), 11 deletions(-)
```

3. **Word:** JOD does not directly generate individual word scripts but it is easy to define a simple utility that does. `pwf`[18] writes individual JOD put dictionary word files. `pwf` is stored in the `utils` dictionary.

```
pwf=: 3 : 0

NB.*pwf v-- write put dictionary words as script files.
NB.
NB. monad:  pwf clPattern
NB.
NB.   pwf 're'  NB. write put dictionary words with prefix 're'
NB.   pwf ''    NB. write all put dictionary words
NB.
NB. dyad:   clPath pwf clPattern
NB.
NB.   'c:/temp' pwf 'de' NB. write to given directory

'' pwf y
:
NB. JOD references !(*)=. dnl get badrc_ajod_ ok_ajod_
NB. !(*)=. isempty_ajod_ jpathsep_ajod_ makedir_ajod_ write_ajod_
pk=.  >@{
tsl=. ] , ('\'"_ = {:) }. '\'"_
if.     badrc_ajod_ ws=. 0 _1 dnl y        do. ws return.
elseif. badrc_ajod_ ws=. 0 10 get 1 pk ws  do. ws return.
NB. individual word scripts using short description text for tacits
elseif. badrc_ajod_ ws=. 0 0 1 wttext__MK__JODobj 1 pk ws  do. ws return.
elseif.do.
  try.
    NB. if (x) path is empty use put dictionary directory (alien\words)
    if. isempty_ajod_ x do.
      DL=. {:{.DPATH__ST__JODobj NB. !(*)=. DL
      NB. insure subdirectory when (x) is empty
      NB. when (x) is nonempty assume it exists
```

[18] `pwf` is not a complete solution to exporting individual JOD objects as scripts. It only exports words and ignores tests, groups, macros and other objects. It does not address the issue of synchronizing exported objects with dictionary state. For example, dictionary word deletions are not propagated. If you wish to track dictionary state use the **`Dictionary`** (71) level method.

```
      makedir_ajod_ <jpathsep_ajod_ tsl x=. ALI__DL,'words'
    end.
    NB. write individual word files
    ws=. 1 pk ws
    wpf=. (<jpathsep_ajod_ tsl x) ,&.> (0 {"1 ws) ,&.> <'.ijs'
    ok_ajod_ wpf [ (toHOST&.> 1 {"1 ws) write_ajod_&.> wpf
  catchd. jderr_ajod_ 'unable to write all word file(s)'
  end.
end.
)
```

Using `pwf` is a simple matter of getting and running it. The following exports `joddev` words to `joddev/alien/words` and then commits differences in `Git`.

```
   NB. Step 1: J session commands - open dictionaries
   od ;:'joddev jod utils' [ 3 od ''
+-+--------------------+------+---+-----+
|1|opened (rw/ro/ro) ->|joddev|jod|utils|
+-+--------------------+------+---+-----+

   NB. load (pwf, showpass) into the (ijod) locale
   'ijod' get ;:'pwf showpass'
+-+-----------------+
|1|2 word(s) defined|
+-+-----------------+

   NB. edit/modify/create words and save changes ...

   #showpass pwf ''  NB. write and count word files
+-+-----------------------------------+...
|1|c:/jod/joddev/alien/words/ASCII85.ijs|...
+-+-----------------------------------+...
121
```

```
$ echo Step 2: Bash shell commands > /dev/null

bakerjd99@NINJA /c/jod/joddev/alien/words (master)
$ pwd
/c/jod/joddev/alien/words

bakerjd99@NINJA /c/jod/joddev/alien/words (master)
$ git status -s
 M pwf.ijs

bakerjd99@NINJA /c/jod/joddev/alien/words (master)
$ git add pwf.ijs

bakerjd99@NINJA /c/jod/smugdev/script (master)
```

```
$ git commit -m '(pwf) comments added'
[master e93fga4] (pwf) comments added
 1 files changed, 3 insertions(+), 2 deletions(-)
```

J Hungarian Notation for J

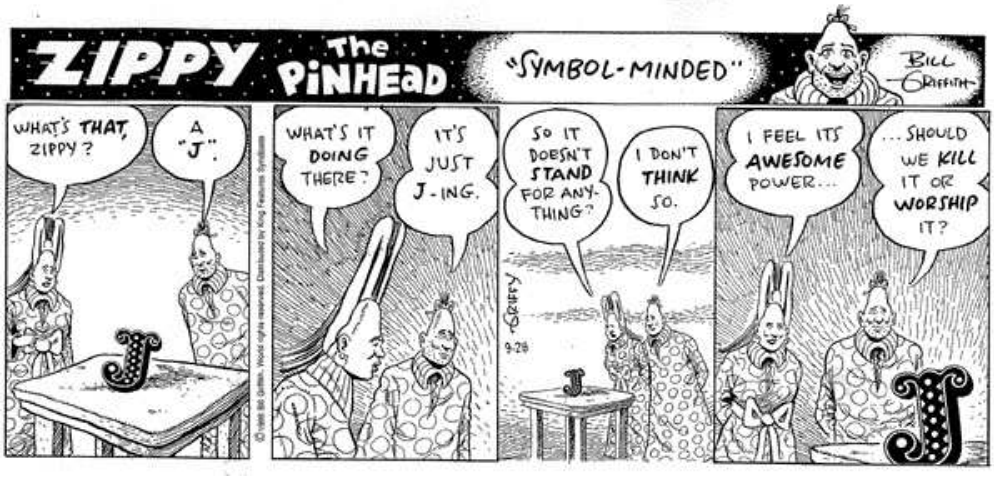


Figure 5: Zippy [2] isn't the only one challenged by the *awesome power* of J.

J.1 Whither Hungarian

J is a *dynamically typed* language! What this means is that you do not have to declare the types of arguments and that types can change during program execution. Discarding the type declaration machinery found in other programming languages simplifies J coding but it can impose its own problems. Without declarations it's not always clear *what is a valid argument*. J does not require that you provide hints and, in J's *tacit* case, it does not even require that you provide arguments! Given the language's terse nature this quickly leads to an incomprehensible style that J detractors have dubbed *line noise*.

To distinguish my J code from line noise I have adapted a documentation style known as Hungarian notation [15]. Hungarian notation inspires devotion and disgust. Many swear by it and many swear at it. For me a convention is worthwhile if it *helps me* understand code. The style outlined here helps me understand and maintain J code. It might help you too.

J.2 J Noun Types

There are two broad classes of arguments in J: nouns and verbs. Nouns are data; they correspond to arguments found in other programming languages. Verbs are programs. J adverbs and conjunctions take verb arguments.[19] Adverbs and conjunctions roughly correspond to the higher order functions

[19] Adverbs and conjunctions also take noun arguments.

found in languages like LISP [11] and Scheme [13]. J *explicit* definition syntax reserves the characters `x y m n u v` for arguments: see Table 13 on page 77.[20] The Hungarian notation described here focuses on noun arguments, (`x` and `y`), because they are the most common.

J Explicit Arguments	
x	*left verb noun argument*
y	*right verb noun argument*
m	*left conjunction noun argument*
n	*right conjunction noun argument*
u	*left adverb/conjunction verb argument*
v	*right conjunction verb argument*

Table 13: Characters reserved for J *explicit* definition arguments. *Tacit* definitions do not directly refer to arguments.

To succinctly describe a J noun you need to be mindful of:

- Type
- Rank
- Boxing

J types are congruent to simple types in other languages. The standard J utility verb `datatype` enumerates primitive J noun types.

```
datatype=: 3 : 0
  n=. 1 2 4 8 16 32 64 128 1024 2048 4096 8192 16384 32768 65536 131072
  t=. '/boolean/literal/integer/floating/complex/boxed/extended/rational'
  t=. t,'/sparse boolean/sparse literal/sparse integer/sparse floating'
  t=. t,'/sparse complex/sparse boxed/symbol/unicode'
  (n i. 3!:0 y) pick <;._1 t
)

  NB. types of list items
  datatype&.> (2x^128);1;'char';(s: ' symbol minded');7 %~ i. 4 5
+--------+-------+-------+------+--------+
|extended|boolean|literal|symbol|floating|
+--------+-------+-------+------+--------+
```

Rank has a precise technical meaning in J but in this context it can be loosely thought of as array dimension. Typical ranks in J are:

[20] Earlier versions of J used `x. y. m. n. u. v.` for arguments. This inflected syntax has been deprecated.

- Single numbers like `42` and characters `'a'` are called atoms. They have *rank 0*.[21]
- Lists like `1 2 3` and `'characters'` correspond to *1-dimensional* arrays in most languages and have *rank 1*.
- Tables like `i. 3 2` are *2-dimensional* arrays and have *rank 2*.
- n dimensional arrays have rank n.

Boxing is structural. J nouns are either boxed or simple. A simple noun has one of the types, (excluding boxed), listed by the `datatype` verb. To mix types in a J array you must box.

```
NB. you must box < to mix types in J arrays
(<u: 'unicode me'),(<i. 2 3),<'types to mix'
```

J.3 Hungarian Noun Descriptions

To describe J nouns I use the following rules:[22]

1. For basic descriptions I use *TypeRank[Name]* where *Type* comes from Figure 6 on page 79, *Rank* is one of:

```
a    atom - rank 0
l    list - rank 1
t    table - rank 2
[n]  general rank n
```

   and *Name* is an optional descriptive name. The *TypeRank* prefix uses the case of Figure 6 on page 79 and *Name* begins with an uppercase letter.

```
paSwitch    boolean (proposition) Switch
ilColors    integer list Colors
ctDocument  character table Document
jt          complex numerix table or matrix
s[3]Xref    rank 3 array of Xref symbols
Rl          extended rational list
bt          boxed table
SClRare     sparse character list Rare
wlPersian   unicode list Persian
ztHolder    empty table Holder
ulWhatever  universal list - any J list
uu          universal universal - any J argument
```

[21] *Rank 0*, or *0-dimensional* objects occur in in all programming languages but are *rarely recognized.* This leads to mountains of ugly special case code. J is more than a programming language; it's a comprehensive and rigorous way to *think* about arrays.

[22] As in the film *Pirates of the Caribbean* these rules are more like *guidelines!*

J Native Type Prefixes		
Prefix	**Native Type**	**(3!:0) code**
p	*boolean*	1
c	*literal*	2
i	*integer*	4
f	*floating*	8
j	*complex*	16
b	*boxed*	32
X	*extended*	64
R	*rational*	128
SP	*sparse boolean*	1024
SC	*sparse literal*	2048
SI	*sparse integer*	4096
SF	*sparse floating*	8192
SJ	*sparse complex*	16384
SB	*sparse boxed*	32768
s	*symbol*	65536
w	*unicode*	131072

(a) J native type prefixes

Generic Prefixes	
Prefix	**Description**
n	*any numeric type including boolean*
N	*any extended numeric type*
u	*universal - any J type*
z	*empty - has at least one 0 axis*

(b) Generic prefixes

Figure 6: Hungarian type prefixes.

2. For boxed nouns of depth one I use a *TypeRankTypeRank[Name]* where the right most pairing describes the boxed types. Boxed nouns of depth one occur often.

```
blcl          boxed list of character lists
blit          boxed list of integer tables
bljtCoord     boxed list of complex tables
blul          boxed list any lists
b[3]s[4]Maps  boxed rank 3 array of rank 4 symbol array Maps
```

3. For more complex nouns, when it's helpful to expose some external structure, I use a mixture of more basic noun descriptions and J syntax.

```
(<blcl),<jtPlane                    two item list
pa;ftXy;<btuu                       three item list
cl;ia;(<blcl),<blcl                 see Table 9 page 60
itYYYYMMDD;slWords;(<bt),<clName    four item list
saRed,saGreen,saBlue                emphasize items of simple noun
```

4. Finally, when more than one description is needed I separate individual descriptions with the *or* symbol ∨ and use as many consecutive lines as required.[23]

[23] The ∨ symbol was chosen because it falls outside of J's ASCII vocabulary and suggests "either-or."

```
dyadic put argument description see page 44
iaObject put clName ∨ blclNames ∨ btNvalues
clLocale put clName ∨ blclNames ∨ btNvalues
(iaObject,iaQualifier) put clName ∨ blclNames
(iaObject,iaQualifier) put clName btNvalues

dyadic dnl argument description see page 21
iaObject dnl zl ∨ clPstr
(iaObject,iaOption) dnl zl ∨ clPstr
(iaObject,iaOption,iaQualifier) dnl zl
(iaObject,iaOption,iaQualifier) dnl clPstr
```

K JOD Mnemonics

*"**M**nemonics **N**eatly **E**liminate **M**an's **O**nly **N**emesis - **I**nsufficient **C**erebral **S**torage."*

jodhelp us!
I get it!
dnl is not just a river in Egypt.
put it where the sun don't shine.
make my day.
globs of gunk.
We're living in a golden jodage.
diddle me this!
grp'ing in the dark.
Am I going to live doc?
It was revolting.
He od'ed man.
et phone home.
It's a brand newd.
He put on a fine display.
Dick uses Jane.
I feel well restd.
All packd up and nowhere to go.

References

[1] Chris Burke. `scriptdoc`, 2008.
J documentation for the `scriptdoc` utility.
http://www.jsoftware.com/help/user/scriptdoc.htm.

[2] Bill Griffith. Official site of Zippy the Pinhead, 2011.
Cartoonists get to the *nub* of cultural matters; they are the J programmers of the visual arts.
http://www.zippythepinhead.com/.

[3] Roger K.W. Hui and Kenneth E. Iverson. *J Dictionary*, 1998.
The *J Dictionary* is the definitive reference for the J programming language. Printed versions of the J Dictionary were produced for early versions of J but now (2011) the J Dictionary exists as a set of `HTML` documents that are distributed with J and maintained online. I yearn for a new printed edition of this magnificent book.
http://www.jsoftware.com/help/dictionary/title.htm.

[4] John D. Baker. `jod` addon, September 2008.
J Wiki download page for the `jod` addon.
http://www.jsoftware.com/jwiki/Addons/general/jod.

[5] John D. Baker. `jodsource` addon, September 2008.
J Wiki download page for the `jodsource` addon.
http://www.jsoftware.com/jwiki/Addons/general/jodsource.

[6] John D. Baker. *The JOD Page*, December 2011.
This website maintains references to all JOD related documents and downloads.
http://bakerjd99.wordpress.com/the-jod-page/.

[7] John D. Baker. `joddocument` addon, June 2012.
J Wiki download page for the `joddocument` addon.
http://www.jsoftware.com/jwiki/Addons/general/joddocument.

[8] Donald Knuth. Personal website, 2008.
The website of the legendary computer scientist Donald Knuth. Knuth created the typesetting language TEX in the 1970's and TEX is still going strong! Genius is hard to replace!
http://www-cs-faculty.stanford.edu/~knuth/.

[9] Oleg Kobchenko. J Graphviz addon, 2008.
J Wiki download page for the `graphviz` addon. After JOD this is my favorite J addon. Oleg Kobchenko has created a jewel for J users!
http://www.jsoftware.com/jwiki/Addons/graphics/graphviz.

[10] Oleg Kobchenko and Chris Burke. JAL J Application Library, 2008.
J Wiki documentation about JAL: the J package manager. JAL is the main tool for downloading and installing J addons.
http://www.jsoftware.com/jwiki/JAL/Package_Manager.

[11] Guy L. Steele. Common lisp the language, 2nd edition, 1990.
Common Lisp is the industrial strength version of the LISP family of programming languages. It's star has been waning since the late 1980's.
http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html.

[12] Git Website. Official Git website, 2012.
The Git version control website.
http://git-scm.com/.

[13] MIT Website. Scheme, 2003.
All about the *Scheme* programming language.
http://www-swiss.ai.mit.edu/projects/scheme/.

[14] Subversion Website. Official Subversion website, 2012.
The Subversion version control website.
http://subversion.tigris.org/.

[15] Wikipedians. Hungarian notation, 2008.
Wikipedia entry for *Hungarian Notation*: it's overwrought but conveys the essentials.
http://en.wikipedia.org/wiki/Hungarian_notation.

List of Tables

List of Figures

Index