



J Object Dictionary

A code, test and documentation database system for J

Author:

John D. Baker

bakerjd99@gmail.com

Release:

1.0.22 - dev

December 11, 2021

Print dimensions: US Letter

Bound printed version at [amazon.com](https://www.amazon.com)

Amazon ASIN key: B08M2KBMND

Document Version History		
Date	Version	Description
December 11, 2021	1.0.22 - dev	current version
October 28, 2020	1.0.2	Amazon edition B08M2KBMND
March 28, 2020	1.0.1	prior pacman release
December 9, 2019	1.0.0	pacman update
August 29, 2019	0.0.999	pacman update
February 14, 2019	0.0.998	pacman update
October 14, 2018	0.9.996	pacman update
...
October 31, 2008	0.8.0	first printed edition
...
March 30, 2008	0.3.5	first draft

Table 1: Abbreviated Document Version History

Contents

1	Introduction	4
1.1	What is JOD?	4
1.2	Why JOD?	4
2	Installing and Configuring JOD	5
3	Quick Tutorial	6
4	Best Practices	11
5	JOD Interface Words	15
5.1	abv — all backup versions	16
5.2	addgrp — add words/tests to group/suite	17
5.3	bget — backup get	17
5.4	bnl — backup name lists	19
5.5	compj — compress J code	20
5.6	del — delete objects	24
5.7	delgrp — remove words/tests from group/suite	25
5.8	did — dictionary identification	25
5.9	disp — display dictionary objects	25
5.10	dnl — dictionary name lists	27
5.11	doc — format comments	28
5.12	dpset — set and change parameters	30
5.13	ed — edit dictionary objects	32
5.14	et — put text into edit window	33
5.15	gdeps — group dependents	33
5.16	get — get objects	33
5.17	getrx — get required to execute	36
5.18	globs — global references	36
5.19	grp — create and modify groups	38
5.20	gt — get edit window text	39
5.21	hlpnl — display short object descriptions	39
5.22	jodage — age of JOD objects	41
5.23	jodhelp — display help	41
5.24	lg — make and load group	42
5.25	locgrp — list groups/suites with word/test	42
5.26	make — generates dictionary scripts	43
5.27	mls — make load script	44
5.28	mn1 — master name lists	45



5.29	<code>newd</code> — create a new dictionary	47
5.30	<code>notgrp</code> — not grouped	48
5.31	<code>nw</code> — edit a new word	48
5.32	<code>nt</code> — edit a new test	49
5.33	<code>od</code> — open dictionaries	50
5.34	<code>packd</code> — backup and pack dictionaries	52
5.35	<code>put</code> — store objects in dictionary	52
5.36	<code>regd</code> — register dictionaries	55
5.37	<code>restd</code> — restore backup dictionaries	56
5.38	<code>revo</code> — list recently revised objects	56
5.39	<code>rm</code> — run macros	57
5.40	<code>rtt</code> — run tautology tests	57
5.41	<code>rxs</code> — regular expression search	58
5.42	<code>uses</code> — return word uses	59
6	JOD Scripts	60
6.1	Generated Script Structure	60
6.2	Dependent Section	61
7	JOD Directory and File Layouts	62
7.1	Master File — <code>jmaster.ijf</code>	62
7.2	Words File — <code>jwords.ijf</code>	62
7.3	Tests File — <code>jtests.ijf</code>	62
7.4	Groups File — <code>jgroups.ijf</code>	66
7.5	Suites File — <code>jsuites.ijf</code>	66
7.6	Macros File — <code>jmacros.ijf</code>	66
7.7	Uses File — <code>juses.ijf</code>	66
A	JOD Distribution	71
B	Building JOD	72
C	JOD Classes	73
D	Reference Path	74
E	JOD Argument Codes	75
F	<code>jodparms.ijs</code>	77
G	<code>jodprofile.ijs</code>	79



H	joduserconfigbak.ijs	83
I	JOD startup.ijs entries	84
J	Turning JOD Dump Script Tricks	85
K	JOD Direct Definition Support	89
L	JOD and Version Control Systems	91
M	Hungarian Notation for J	95
	M.1 Whither Hungarian	95
	M.2 J Noun Types	95
	M.3 Hungarian Noun Descriptions	97
N	JOD Mnemonics	100
	References	101
	List of Tables	103
	List of Figures	103
	Index	104



1 Introduction

1.1 What is JOD?

JOD is a code, test and documentation *refactoring Addon* for the *J programming language* [3] [16]. JOD has been programmed entirely in J¹ 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 Object Dictionary**.

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.

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

²Over-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 `pacman` to install missing addons.

JOD can be installed in two ways.

1. Use `pacman`, J's package manager [1], to download JOD. *Using pacman is the easiest way to install and maintain JOD.*
2. Download the current JOD distribution from [The JOD Page](#) [7] and unzip, preserving directories, to the relative directory⁴ `~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 44. Exercise caution when manually editing JOD's load script section.

³JOD runs on J 8.0x and J 9.0x systems. 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

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

⁵The `'` 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 `pacman`.
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. For macOS and Linux systems use paths like:

JOD	/users/john/jod
JODDUMPS	/users/john/jod/joddumps
JODSOURCE	/users/john/jodtest/labtesting
JODTEST	/users/john/jodtest/test

4. `jodhelp` (41) 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 Addons category: see Figure 2 on page 8. If JOD labs are not listed in the Addons 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:

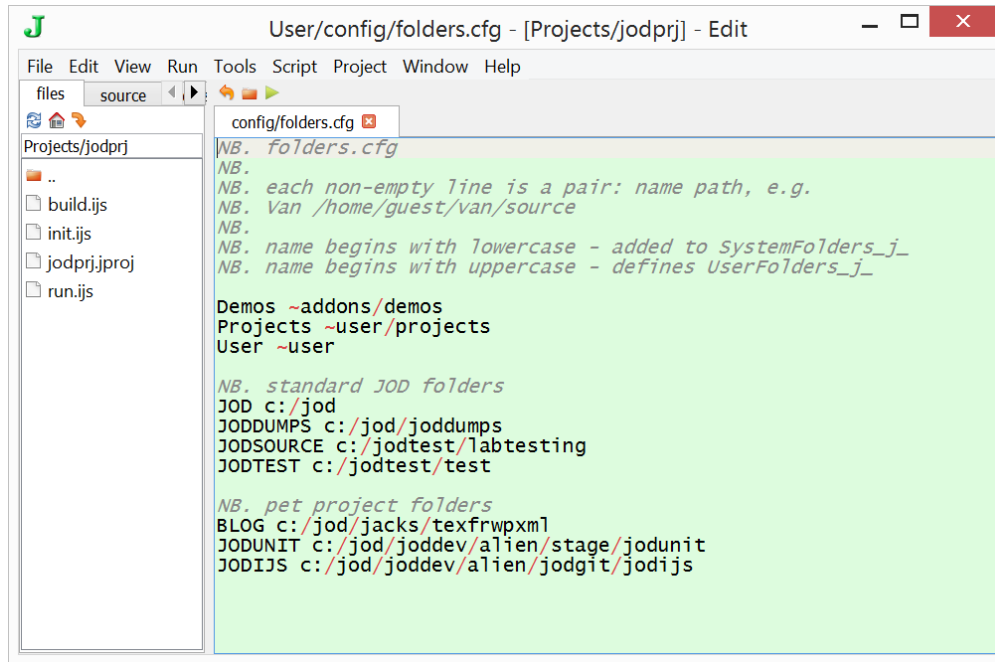
```
load 'general/jod'    NB. start JOD
```

Create dictionaries. To use JOD you must create some dictionaries with `newd`: see page 47. 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.

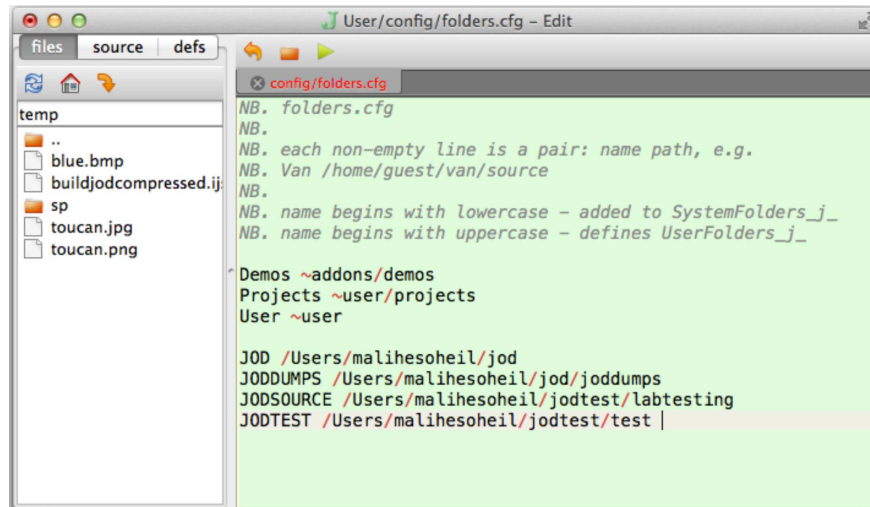
⁶See the blog post [JOD Update: Version 0.9.97*](#) for J configuration advice.

⁷The number “(n)” in JOD lab titles is a suggested order.



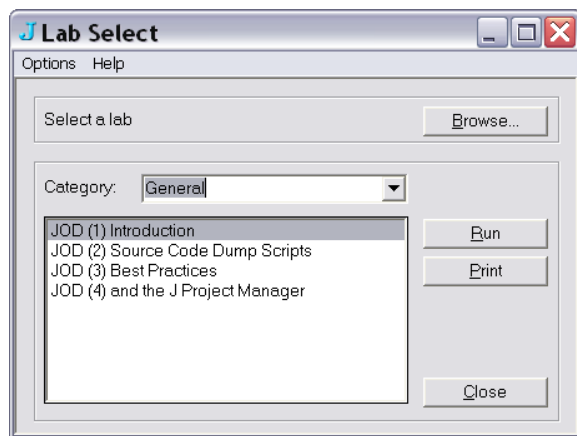


(a) J 9.0x Windows JQT configuration editor

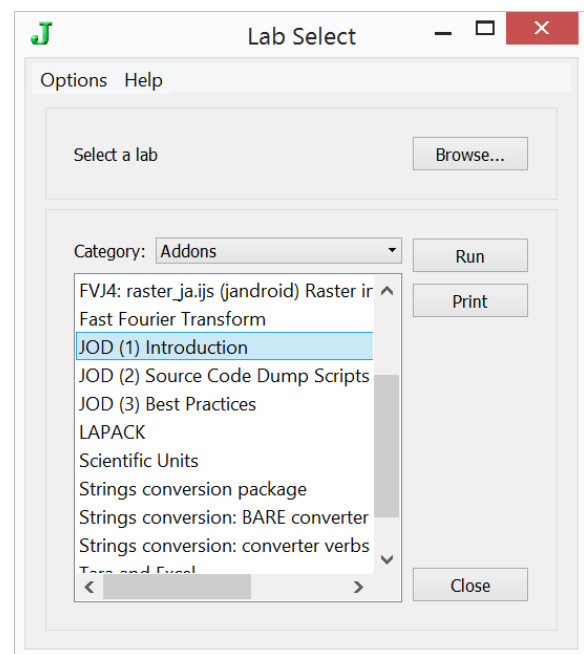


(b) J 8.0x Mac JQT configuration editor

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 8.0x JQT lab dialog

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

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

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

Open dictionaries. To use a dictionary it must be open. Open dictionaries with `od`: see page 50.

```
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 D on page 74. `did`, see page 25, 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 52.

```
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 33, 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 38.

```
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 pacman updates so JOD load scripts are stored in the user's `startup.ijs` file: see [appendix I](#) on page 84.

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



```
+--+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

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

Back up the put dictionary. *You're either backed up or f'ed up—there are no other options!* JOD makes backing up easy: see [packd](#) on page 52.

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

Dump dictionaries on path. [make](#), see page 43, 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. To learn more work through the JOD labs and read the JOD [Jupyter](#) [14] notebooks on Github at: <https://github.com/bakerjd99/jod>.

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 Addons lab category.⁸

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 47. It's also a good idea to define a subdirectory structure that mirrors J's versions.

⁸Prior J versions put JOD labs in the General category.



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'

Back up, back up, and then back up. It's easy to back up with JOD so back up often: see [packd](#) and [restd](#) on pages 52 and 56.

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 43. 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 ''

Dump scripts are essential JOD dictionary maintenance tools: see appendix [J](#) on page 85.

Make a master re-register script. JOD only sees dictionaries registered in the `jmaster.ijf` file: see Table 3 on page 63. Maintaining a list of registered dictionaries is recommended. JOD can generate a re-register script: see [od](#) on page 50. 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 36. References enable many useful operations. References allow [getrx](#), see page 36, 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 28, [put](#) on page 52 and [nw](#) on page 48.

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 48 and [ed](#) on page 32 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 57.

```
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 57 or [rtt](#) on page 57.

```
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 [G](#) on page [79](#). 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` that can be read with `jodhelp`: see page [41](#).

```
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 [M](#) on page [95](#).



When JOD loads it creates a number of locale classes, (see appendix C on page 73), 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.⁹ If any JOD word fails `jodsf` will catch the error and return JOD's standard (`paRc; clMessage`) result.

5.1 *abv* — all backup versions

abv searches *binary* backups and returns lists of *backup names*. *abv* is used with *bget*, see page 17, to recover objects in backups.

Monad: *abv clPstr* *✓ blclBNames*

```
NB. list all backup words from most recent to oldest
abv ''
```

```
NB. list backup words starting well
abv 'well'
```

```
NB. get scary backup words
'rc ncv'=. bget }. abv 'scary'
```

```
NB. edit all backup words starting out
ed > 1 { bget }. abv 'out'
```

Dyad: *iaObject abv clPstr*
(iaObject, iaOption) abv clPstr

⁹Provisions for defining a *non-error-trapping* interface exist in JOD source code.

```

NB. backup smoke tests
1 abv 'smoke'

NB. list backup macro post processors
4 abv 'POST_'

NB. edit all group headers with prefix foo
'rc ncv'=. 2 bget }. 2 abv 'foo'
ed ncv

```

5.2 *addgrp* — add words/tests to group/suite

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

Dyad: *clGroup addgrp clName V blclNames*
(clSuite;iaObject) addgrp clName V 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'

```

5.3 *bget* — backup get

bget fetches objects from either the last dictionary *binary* backup or particular backups. Backups are created by [packd](#), see page 52.

Fetches *objects are not defined in locales* for the simple reason that backup fetches may return many versions of the same object. *bget* like [bnl](#), only searches put dictionary backups and unlike [get](#), (see page 33), *bget* does not fetch name classes, timestamps or byte sizes.

The (y) argument of *bget* is similar to the (y) argument of [get](#). Both take lists of names. *get* takes lists of J names that do not reference locales or objects and *bget* takes *get* names and *backup names* where a backup name is a *get* name that may end with a *backup suffix*. The following *bget* examples illustrate backup names.

Monad: *bget clBName V blclBNames*



NB. get word from last backup

```
bget 'oops'
```

NB. fetch words from last backup

```
bget ;: 'shawn of the dead'
```

NB. fetch words from particular backups - backup names

```
bget <;._1 ' us.12 poor.10 little.08 words.08 lastback'
```

NB. fetch many versions of a word - backup names

```
bget <;._1 ' me me.12 me.09 me.08 me.02'
```

NB. all words from last backup

```
bget }. bnl ''
```

NB. all words from backup #55

```
bget (}. bnl '.55') ,&.> <' .55'
```

Dyad: *iaObject bget clBName ∨ blclBNames ∨ uuIgnore
(iaObject,iaOption) bget clBName ∨ blclNames*

```
5 bget ''      NB. dictionary document from last backup
```

```
5 bget '.12'   NB. dictionary document from particular backup
```

```
5 bget }. bnl '.' NB. documents from all backups
```

NB. fetch a suite header from a particular backup

```
3 bget 'sweet.04'
```

NB. fetch a particular suite test list

```
3 1 bget 'sweet.04'
```

NB. three versions of a group's header - similar to (get)

NB. where (2 get 'group') returns the group header

```
2 bget <;._1 ' gfoo.12 gfoo.05 gfoo.00'
```

NB. three versions of a group's word list - backup names

```
2 1 bget <;._1 ' gfoo.12 gfoo.05 gfoo.00'
```

NB. fetch many versions of a macro - backup names

```
4 bget <;._1 ' mymacro.13 mymacro.09 mymacro.07 mymacro.02'
```



5.4 *bnl* — backup name lists

bnl searches *put dictionary* backups, see page 10 and appendix D on page 74, and returns backup name lists. *bnl*'s arguments are similar to *dnl*, see page 27.

Monad: *bnl* *z1* *✓* *clPstr*

NB. list all put dictionary backup numbers

bnl *'.'*

NB. words in last put dictionary backup

bnl *''*

NB. all words in all backups

bnl&> *}*. *bnl* *'.'*

NB. words in last backup matching prefix

bnl *'prefix'*

NB. words in backup #42 matching prefix

bnl *'prefix.42'*

Dyad: *iaObject* *bnl* *z1* *✓* *clPstr*

(iaObject,iaOption) *bnl* *z1* *✓* *clPstr*

(iaObject,iaOption,iaQualifier) *bnl* *z1*

(iaObject,iaOption,iaQualifier) *bnl* *clPstr*

0 *bnl* *''* *NB. all words in last backup (monad)*

0 *bnl* *'..11'* *NB. all words in backup #11*

1 *bnl* *''* *NB. all tests in last backup*

1 *bnl* *'..13'* *NB. all tests in backup #13*

2 *bnl* *''* *NB. all groups in last backup*

2 *bnl* *'..7'* *NB. all groups in backup #7*

NB. suites in backup #14 beginning with boo

3 *bnl* *'boo.14'*

NB. words in last backup containing the string str

0 2 *bnl* *'str'*



NB. words in backup #6 containg the string ass

```
0 2 bnl 'ass.6'
```

NB. words in all backups containing yada

```
0 2&bnl&> (<'yada') ,&.> }. bnl '.'
```

NB. word names in last backup ending with string str

```
0 3 bnl 'str'
```

NB. adverbs in backup #17 beginning with str

```
0 1 1 bnl 'str.17'
```

NB. verb names in last backup containing str

```
0 2 3 bnl 'str'
```

NB. nouns in backup #9 ending with str

```
0 3 0 bnl 'str.9'
```

NB. nouns in all backups ending with EX

```
0 3 0&bnl&> (<'EX') ,&.> }. bnl '.'
```

NB. J macro names beginning with jscript

```
4 1 21 bnl 'jscript'
```

NB. MARKDOWN macro names in backup #18 containing mark

```
4 2 27 bnl 'mark.18'
```

NB. TEXT macro names in backup #23 ending badly

```
4 3 25 bnl 'badly.23'
```

NB. put dictionary backup numbers and file timestamps

```
14 bnl '.'
```

5.5 *compj* — compress J code

compj compresses J code by removing comments, white space and shortening safe local identifiers to single characters.¹⁰ Code compression is useful when preparing production scripts. The JOD system script:

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

¹⁰If more than one character is required to rename all local identifiers *compj* uses a letter prefixed high base numbering scheme.



is an example of a compressed J script. In its fully commented form this script is about 230 kilobytes when squeezed with `compj` it shrinks to about 84 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 36, to expose any name problems.

Ambiguous names in J are words created in object instances, temporary locale globals, names masked by indirect assignments and objects created with `execute`. When you use ambiguous names augment your code with sufficient information to clearly resolve and cross reference all names.

JOD provides two comment scope tags `(*)=.` and `(*)=:` to clarify ambiguous names. 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
locoref=. ;:'we are hidden locals'
(locoref)=. 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
)
```

compj recognizes two additional comment tags *(-.)=:* and *(/:)=:*. The first limits compression to only white space removal and the second *obfuscates* local names.

1. *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.
)
```

2. *compj* will obfuscate local names when explicit words are marked with the comment tag *(/:)=:*. Name obfuscation makes code difficult to read and should only be used when words are running general scripts with 0! :k foreigners. *(-.)=:* overrides *(/:)=:*.



```
obfuscateme=: 3 : 0
```

```
NB. I run a script that may reassign my names.
```

```
NB. Obfuscation makes this unlikely.
```

```
NB. (/:)=: do not eschew obfuscation
```

```
script=. y
```

```
NB. running (script) may clash with local (script)
```

```
0!:000 y
```

```
'bytes run' , ":# script
```

```
)
```

More examples of the use of comment 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 [jodijis.zip](#) from [The JOD Page](#) [7]. JOD source is distributed as JOD dictionary dump scripts and as fully commented class files in [jodijis.zip](#).

Monad: *compj clName V 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 V blclNames*

```
NB. remove comments preserving leading whitespace and
```

```
NB. original identifiers - useful for reading code
```

```
NB. without distracting comments
```

```
>1{ 1 compj 'justthecode'
```



5.6 *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.¹¹ Deleting a word that belongs to a group breaks the group: similarly for suites. An attempt to *get* or *make* a broken group or suite will result in an error. You can recover from this error by deleting references, (see below), and regrouping.

Monad: *del clName V blclNames*

NB. delete one word

```
del 'word'
```

NB. delete many words

```
del 'go'; 'ahead'; 'delete'; 'us'
```

Dyad: *iaObject del clName V 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'
```

¹¹In database terms *del* can violate *referential integrity*. Early versions of JOD maintained referential integrity but this proved cumbersome and was dropped.

5.7 **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 V blclNames*
(clSuite;iaObject) delgrp clName V 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'

5.8 **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

5.9 **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 V blclNames*



NB. display a word

```
disp 'word'
```

NB. display many words

```
disp ;:'go ahead show us'
```

Dyad: *iaObject disp clName V blclNames*
 (*iaObject,iaOption*) *disp clName V 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 ''
```



5.10 *dn1* — dictionary name lists

dn1 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 *dn1* list. *dn1* arguments follow the pattern:

```
(n, ⟨p, ⟨d⟩⟩) dn1 '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: *dn1* *z1* *V* *clPstr*

NB. list all words on current dictionary path
dn1 ''

NB. list all words that begin with prefix
dn1 'prefix'

Dyad: *iaObject dn1 z1 V clPstr*
(iaObject, iaOption) dn1 z1 V clPstr
(iaObject, iaOption, iaQualifier) dn1 z1
(iaObject, iaOption, iaQualifier) dn1 clPstr

```
0 dn1 '' NB. all words (monad)
1 dn1 '' NB. list all tests
2 dn1 '' NB. list all groups
3 dn1 '' NB. list all suites
4 dn1 '' 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 *dn1* 'str'

NB. match word names containing the string str
0 2 *dn1* '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'
```

5.11 *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` 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'
```

5.12 **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 F, on page 77, 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 z1 V clName V (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
```

NB. retain age of objects in dump scripts

```
dpset 'RETAINAGE';1
```

NB. set J configured make script directory

```
dpset 'ROOTFOLDER'; '~USERROOT/mlsscripts/'
```



5.13 *ed* — edit dictionary objects

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

Monad: *ed* *clName* *∨* *blclNames* *∨* *btNameClassValue*

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

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

NB. edit objects in (name,[class],value) tables
 'rc ncv'=. 4 get }. 4 1 21 dnl 'build'
ed ncv

NB. edit backup versions of a word in a single edit window
 'rc ncv'=. bget <;._1 ' me.12 me.09 me.08 me.02'
ed ncv

NB. edit all words in last backup in a single edit window
 'rc ncv'=. bget }. bnl ''
ed ncv

Dyad: *iaObject ed clName ∨ blclNames ∨ btNameClassValue*
(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'



5.14 *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'

5.15 *gdeps* — group dependents

gdeps returns lists of global names in the dependent section of group and suite headers: see page 61.

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'

5.16 *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 52. 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 V blclNames V uuIgnore*
clLocale get clName V 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'
```

NB. get word names and timestamps

```
0 _14 get ;:'our time stamps'
```

NB. get macro names and timestamps

```
4 _14 get ;:'macro creation lastput'
```

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 ''
```

5.17 *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'
```

5.18 *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.

Finally, `globs` expects all assigned local and global word names to differ. If `globs` finds names that are both globally and locally assigned it returns a mixed assignments error.

```
mixedassignment=: 3 : 0
```

*NB. I have a mixed global and local assignment. Usually, this is
NB. poor name choice but there are good reasons for such assignments.*

```
names=. ;:'we are going to be globals'
(names)=: <'clandestine global'
)
```

To override `globs` mixed assignment handling use the comment tag `(<:)=:`. If this tag is used anywhere in a word definition `globs` will ignore any mixed assignments in the tagged word.

```
confusing=: 3 : 0
```

```
NB. allow mixed assignments (<:)=:
scopeless=. <'cis_scoping_is_so_un_woke'
(scopeless)=: scopeless
)
```



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'

5.19 *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 z1 √ 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 z1 V clName V 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'

5.20 *gt* — get edit window text

Fetch text from edit window.

Monad: *gt z1 V 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'

5.21 *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 V 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 V 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'

```



5.22 *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.f` floating point format.¹²

Monad: *jodage clWord V blclWords*

NB. show age of (jodage)

jodage 'jodage'

NB. age of all group words

jodage }. grp 'bstats'

Dyad: *ia jodage clWord V 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 ''

5.23 *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 `pacman` 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*

¹²JOD 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!'

5.24 *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 *numutils* the associated postprocessor is named *POST_numutils*. The prefix *POST_* labels J macro scripts as postprocessors.¹³ 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'

5.25 *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.

¹³A 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'

5.26 *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 60.

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 V 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 V clName V 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 47.

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'
```

5.27 *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.¹⁴

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 60.

¹⁴On 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'

NB. redirect generated script by setting
NB. ROOTFOLDER to a configured J folder
jpath '~user/jodroot'
dpset 'ROOTFOLDER'; '~user/jodroot'
mls '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'

```

5.28 *mn1* — master name lists

mn1 searches the name lists of all registered dictionaries, see [regd](#) on page 55. *mn1* does not require open dictionaries. *mn1*'s arguments mirror [dn1](#) with the exception of negative option codes, see page 27. The primary use of *mn1* is looking for duplicate names. Duplicate names limit the utility of words and make it difficult to use dictionaries on arbitrary paths. If the dictionaries *d0*, *d1* and *d2* contain common names the paths resulting from the following commands are not necessarily the same.



```
od ;:'d0 d1 d2'
od ;:'d2 d0 d1'
```

Monad: *mn1 z1 V clPstr*

NB. list all words in all registered dictionaries
mn1 ''

NB. list all words in all dictionaries that begin with prefix
mn1 'prefix'

Dyad: *iaObject mn1 z1 V clPstr*
(iaObject,iaOption) mn1 z1 V clPstr
(iaObject,iaOption,iaQualifier) mn1 z1
(iaObject,iaOption,iaQualifier) mn1 clPstr

```
0 mn1 '' NB. all words in all dictionaries (monad)
1 mn1 '' NB. all tests in all dictionaries
2 mn1 '' NB. all groups in all dictionaries
3 mn1 '' NB. all suites in all dictionaries
4 mn1 '' NB. all macros in all dictionaries
```

NB. match word names beginning with str
0 1 mn1 'str'

NB. match word names containing the string str
0 2 mn1 'str'

NB. match word names ending with string str
0 3 mn1 '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 mn1 'str'

NB. verb names containing str
0 2 3 mn1 'str'



NB. nouns ending with str

```
0 3 0 mnl 'str'
```

NB. J macro names beginning with jscript

```
4 1 21 mnl 'jscript'
```

NB. LaTeX macro names containing latex

```
4 2 22 mnl 'latex'
```

NB. HTML macro names ending with html

```
4 3 23 mnl 'html'
```

A negative second item option code returns duplicate names.

NB. duplicate words containing str

```
0 _2 3 mnl 'str'
```

NB. duplicate group names beginning with so

```
2 _1 mnl 'so'
```

NB. duplicate suite names ending with str

```
3 _3 mnl 'str'
```

5.29 *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 66. 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 ...'
```

5.30 *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* \vee *blclNames*

NB. recent ungrouped words

```
notgrp }. revo ''
```

NB. all ungrouped words

```
notgrp }. dnl ''
```

Dyad: *iaOption notgrp clName* \vee *blclNames*

NB. ungrouped words (monad)

```
2 notgrp }. dnl ''
```

NB. tests that are not in suites

```
3 notgrp }. 1 dnl ''
```

5.31 *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.¹⁵

Monad: *nw clName*

¹⁵Key combinations like CTRL-W that run editor scripts depend on the J platform.

```

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 explicit adverb
1 nw 'adverb'

NB. edit an explicit 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'

```

5.32 *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*, (yyymondd 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'

```

5.33 *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 z1* \vee *clDictionary* \vee *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 z1* \vee *clDictionary* \vee *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 ''*



5.34 *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*. Individual objects in backups can be retrieved with *bget*, see page 17. 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.¹⁶

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'

5.35 *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 33.

Monad: *put clName V blclNames*

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

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

¹⁶There 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 setup.

```
Dyad:  iaObject put clName V blclNames
        iaObject put clDocument V btNvalues
        clLocale put clName V blclNames V btNvalues
        (iaObject,iaQualifier) put clName V blclNames
        (iaObject,iaQualifier) put clName V 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 precedes the code generated by [make](#) on page 43.

```
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'
```

Finally put can store timestamps in the form returned by *get*.

```
NB. push yyyyymmdd.fid lastput dates 15 years forward
'rc nts'=: 0 _14 get }. dnl 'new'
0 _14 put (<0 150000 + "0 1 ;1{nts) 1} nts
```

5.36 *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 62. 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
```



5.37 *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'
```

5.38 *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 z1 V clName*

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

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

Dyad: *iaObject revo z1 V clName*

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

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



5.39 *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 V blclNames*

NB. run J macro
rm 'macro'

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

Dyad: *iaOption rm zl V clName V blclNames*

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

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

5.40 *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 foreigners 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 V 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 V 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'
```

5.41 *rxs* — regular expression search

rxs searches dictionary *text objects* with regular expressions. Text objects are essentially all nonnouns. Typical text objects are verbs, test cases, macros, group and suite headers, long and short documentation texts, and dictionary documentation. *rxs*'s arguments use JOD object and qualifier codes: see appendix E on page 75.

Monad: *rxs* blclNames

NB. all nonempty nonnoun search text

```
rxs }. dnl ''
```

NB. nouns are excluded from searches

```
rxs }. 0 1 0 dnl ''
```

Dyad: clRegex *rxs* blclNames

```
(clRegex ; iaOption) rxs z1 V blclNames
```

```
(clRegex ; ilOptions) rxs z1 V blclNames
```

NB. empty patterns return nonempty nonnoun search text

```
'' rxs }. dnl''
```

NB. nonempty group and suite headers

```
('';2) rxs }. 2 dnl ''
```

```
('';3) rxs }. 3 dnl ''
```



NB. first match of *unique* in nonnouns with names starting with *re*
`'unique' rxs }. dnl 're'`

NB. first match of *catch.* in path words - note pattern escape
`('catch\.';0 7 1) rxs }. dnl ''`

NB. words in group *mytheory* containing *NIMP*:
`'NIMP:' rxs }. grp 'mytheory'`

NB. all *J* quoted text in nonnoun path words starting with *s*
`(''(''''|[''''])*''';0 7 2) rxs }. dnl 's'`

NB. all digits in nonempty long word document text
`('[[[:digit:]]+';0 9 2) rxs }. dnl ''`

NB. digits in long word document text with match position arrays
`('[[[:digit:]]+';0 9 3) rxs }. dnl ''`

NB. many useful *J* regex patterns are in the *jregex* locale

NB. all *J* quoted text with match position arrays
`(Jchar_jregex;0 7 3) rxs }. dnl 's'`

NB. all *J* names in nonempty group header text
`(Jname_jregex;2 7 2) rxs }. 2 dnl ''`

NB. first match of *github* in dictionary document
`('github';5) rxs ''`

NB. all matches of *github* in dictionary document with position arrays
`('github';5 7 3) rxs ''`

5.42 *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 V clName`

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

Dyad: `iaObject uses blclName V 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 43, 44, 42), 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 61.



Generated Script Structure			
Section	Type	Description	Example
Setup	Active	Define group and Suite headers. Headers may contain one dependent section: see page 61.	<i>NB. define a group header</i> 2 1 put 'groupname';' ... script text ... ' <i>NB. define a suite header</i> 3 1 put 'suite name';' ... suite text ... '
Load	Passive	Load lists of words or tests. <i>Only word lists are passive.</i> Tests are typically active scripts.	<i>NB. form group from stored words</i> grp 'groupname' ; ;:'words in group' <i>NB. form suite from stored tests</i> 3 grp 'suite name' ; ;:'stored tests'
Start	Active	Associate a postprocessor macro with a group or suite. Postprocessors are prefixed with POST_	<i>NB. group postprocessor</i> 4 put 'POST_groupname';21;' ... script ... ' <i>NB. test suite postprocessor</i> 4 put 'POST_suite name';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 38), 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 43, 44 and 42.* 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 33.

```

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.ijs`, see Table 3 on page 63, and creates a set of standard directories: see Figure 3 on page 66. This section describes the internal structure of JOD's binary `jfiles`.

7.1 Master File — `jmaster.ijs`

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

`jmaster.ijs` is an index of currently registered dictionaries and standard dictionary meta-data. The component layout of `jmaster.ijs` is given in Table 3 on page 63.

7.2 Words File — `jwords.ijs`

`jwords.ijs` is a binary component `jfile`. `jwords.ijs` contains word definitions and meta-data. The component layout of `jwords.ijs` is given in Table 4 on page 64.

7.3 Tests File — `jtests.ijs`

`jtests.ijs` is a binary component `jfile`. `jtests.ijs` contains test definitions and meta-data. The component layout of `jtests.ijs` is given in Table 5 on page 65.



jmaster.ijf		
Component	Hungarian	Description
<i>c</i> 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.
<i>c</i> ₁	(cl;i,X)	Version m.m.p character, build count and unique master file id.
<i>c</i> ₂	bt	Dictionary names, numbers, directories and read-write status. When a dictionary is opened for update (READWRITE default) by od (pp. 50) 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.
<i>c</i> ₃	bt	Previous master directory. Essentially a copy of component two less at most one deleted or new dictionary.
<i>c</i> ₄ → <i>c</i> ₆		Reserved.
<i>c</i> ₇	bt	Active dictionary parameters. 0 { - b1cl ; parameter names 1 { - b1cl ; short parameter explanation 2 { - bluu ; default values
<i>c</i> ₈	bt	Copy of active dictionary parameters.
<i>c</i> ₉	bt	Default dictionary parameters.
<i>c</i> 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
<i>c</i> ₀	blnl	Length and last directory change.
<i>c</i> ₁	fl	Pack count and last backup or restore timestamp. Pack count prefixes backup and dump files.
<i>c</i> ₂	cl	Dictionary documentation newd , regd . (pp. 47,55)
<i>c</i> 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. 63). 0 { cl ; parameter 1 { uu ; value
		Main inverted items, <i>c</i> ₄ → <i>c</i> ₁₁ have the same length.
<i>c</i> ₄	blcl	Word list (main index 1).
<i>c</i> ₅	il	Word components (main index 2).
<i>c</i> ₆	il	Name class list.
<i>c</i> ₇	fl	Last put date list <i>yyyymmdd.f</i> d (fractional day).
<i>c</i> ₈	fl	Creation put list <i>yyyymmdd.f</i> d (fractional day).
<i>c</i> ₉	il	Word size in bytes.
<i>c</i> ₁₀		Reserved.
<i>c</i> ₁₁	blcl	Short word explanations.
<i>c</i> ₁₂ → <i>c</i> ₃₈		Reserved.
		The remaining component pairs contain word data. The word names match the entries in the word index list.
<i>c</i> ₃₉	bluu	Word definition. 0 { - cl ; word name 1 { - ia ; name class 2 { - cl ✓ uu ; word value: nouns binary, all others character lists
<i>c</i> ₄₀	bluu	Word documentation and other. 0 { - cl ; word name 1 { - uu ; unused - reserved 2 { - uu ; unused - reserved 3 { - cl ; text documentation
<i>c</i> ₄₁	bluu	Like <i>c</i> ₃₉
<i>c</i> ₄₂	bluu	Like <i>c</i> ₄₀
...
<i>c</i> _{<i>n</i>}	...	Like <i>c</i> _{<i>n</i>-2}

Table 4: *jwords.ijf* file component layout

jtests.ijf		
Component	Hungarian	Description
<i>c</i> ₀	blnl	Length and last directory change.
<i>c</i> ₁ → <i>c</i> ₃		Reserved.
		Main inverted items, <i>c</i> ₄ → <i>c</i> ₁₁ have the same length.
<i>c</i> ₄	blcl	Test list (main index 1).
<i>c</i> ₅	il	Test components (main index 2).
<i>c</i> ₆		Reserved to match <i>jwords.ijf</i> .
<i>c</i> ₇	fl	Last put date list <i>yyyymmdd.fd</i> (fractional day).
<i>c</i> ₈	fl	Creation put list <i>yyyymmdd.fd</i> (fractional day).
<i>c</i> ₉	il	Test size in bytes.
<i>c</i> ₁₀		Reserved.
<i>c</i> ₁₁	blcl	Short test explanations.
<i>c</i> ₁₂ → <i>c</i> ₃₈		Reserved.
		The remaining component pairs contain test data. The test names match the entries in the test index <i>c</i> ₄ list.
<i>c</i> ₃₉	blcl	Test definition. 0 { - cl ; test name 1 { - cl ; test value
<i>c</i> ₄₀	bluu	Test documentation and other. 0 { - cl ; test name 1 { - uu ; unused - reserved 2 { - uu ; unused - reserved 3 { - cl ; text documentation
<i>c</i> ₄₁	blcl	Like <i>c</i> ₃₉
<i>c</i> ₄₂	bluu	Like <i>c</i> ₄₀
...
<i>c</i> _{<i>n</i>}	...	Like <i>c</i> _{<i>n</i>-2}

Table 5: *jtests.ijf* file component layout

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

`.../name` The dictionary root directory holds binary jfiles, see section 7 on page 62.

- `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` (52).
- `name/document` Unused — reserved for documents related to this dictionary.
- `name/dump` J dump script files generated by `make` (43).
- `name/script` J script files generated by `make`, `mls` and `lg` (43, 44, 42).
- `name/suite` J test scripts generated by `make` (43).

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.

7.4 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 67.

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 68.

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 69.

7.7 Uses File — *juses.ijf*

juses.ijf is a binary component jfile. *juses.ijf* contains word references: see `globs` subsection 5.18, on page 36.



jgroups.ijf		
Component	Hungarian	Description
<i>c</i> ₀	blnl	Group count and last directory change.
<i>c</i> ₁ → <i>c</i> ₃		Reserved.
		Main inverted items, <i>c</i> ₄ → <i>c</i> ₁₁ have the same length.
<i>c</i> ₄	blcl	Group list (main index 1).
<i>c</i> ₅	il	Group components (main index 2).
<i>c</i> ₆		Reserved to match <i>jwords.ijf</i> .
<i>c</i> ₇	fl	Last put date list <i>yyyymmdd.fd</i> (fractional day).
<i>c</i> ₈	fl	Creation put list <i>yyyymmdd.fd</i> (fractional day).
<i>c</i> ₉ → <i>c</i> ₁₀		Reserved.
<i>c</i> ₁₁	blcl	Short group explanations.
<i>c</i> ₁₂ → <i>c</i> ₃₈		Reserved.
		The remaining component pairs contain group data. The group names match the entries in the group index <i>c</i> ₄ list.
<i>c</i> ₃₉	bluu	Group definition. 0 { - cl ; group name 1 { - cl ; group prefix script 2 { - blcl ; group content list
<i>c</i> ₄₀	bluu	Group documentation and other. 0 { - cl ; group name 1 { - uu ; unused - reserved 2 { - uu ; unused - reserved 3 { - cl ; text documentation
<i>c</i> ₄₁	bluu	Like <i>c</i> ₃₉
<i>c</i> ₄₂	bluu	Like <i>c</i> ₄₀
...
<i>c</i> _{<i>n</i>}	...	Like <i>c</i> _{<i>n</i>-2}

Table 6: *jgroups.ijf* file component layout

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 <i>jwords.ijf</i> .
c_7	fl	Last put date list <i>yyyymmdd.fd</i> (fractional day).
c_8	fl	Creation put list <i>yyyymmdd.fd</i> (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

jmacros.ijf		
Component	Hungarian	Description
<i>c</i> ₀	blnl	Macro count and last directory change.
<i>c</i> ₁ → <i>c</i> ₃		Reserved.
		Main inverted items, <i>c</i> ₄ → <i>c</i> ₁₁ have the same length.
<i>c</i> ₄	blcl	Macro list (main index 1).
<i>c</i> ₅	il	Macro components (main index 2).
<i>c</i> ₆		Reserved to match <i>jwords.ijf</i> .
<i>c</i> ₇	fl	Last put date list <i>yyyymmdd.fd</i> (fractional day).
<i>c</i> ₈	fl	Creation put list <i>yyyymmdd.fd</i> (fractional day).
<i>c</i> ₉	fl	Macro size in bytes.
<i>c</i> ₁₀		Reserved.
<i>c</i> ₁₁	blcl	Short macro explanations.
<i>c</i> ₁₂ → <i>c</i> ₃₈		Reserved.
		The remaining component pairs contain macro data. The macro names match the entries in the macro index <i>c</i> ₄ list.
<i>c</i> ₃₉	blcl	Macro definition. 0 { - <i>c</i> ₁ ; macro name 1 { - <i>c</i> ₁ ; macro script
<i>c</i> ₄₀	bluu	Macro documentation and other. 0 { - <i>c</i> ₁ ; macro name 1 { - <i>uu</i> ; unused - reserved 2 { - <i>uu</i> ; unused - reserved 3 { - <i>c</i> ₁ ; text documentation
<i>c</i> ₄₁	blcl	Like <i>c</i> ₃₉
<i>c</i> ₄₂	bluu	Like <i>c</i> ₄₀
...
<i>c</i> _{<i>n</i>}	...	Like <i>c</i> _{<i>n</i>-2}

Table 8: *jmacros.ijf* file component layout

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 <i>jwords.ijf</i> but occupies the same component range for packd (pp. 52). 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 `pacman` the *J package manager* [10]. The JOD distribution is broken into three `pacman` 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` [6]: this package contains JOD PDF documents. Installing this package places these documents on local drives for `jodhelp`, see page 41.



B Building JOD

JOD is an open-source system. Anyone is free to examine and modify JOD source code. All JOD source is stored in JOD development dictionaries. Installing the `jodsource` addon makes this code available.

JOD dictionaries also contain utilities for building and distributing JOD.

```
NB. open JOD development dictionaries
od ;:'joddev jod utils' [ 3 od ''
```

```
NB. build JOD
rm 'buildjoddistribution'
```

`buildjoddistribution` extracts JOD source code from development dictionaries and generates the compressed or *minimized* scripts used by the JOD addon.

```
NB.*buildjoddistribution s-- full JOD distribution build.
```

```
NB. create temporary working locale
cocurrent 'base'
coclass tmploc_AAajodbuild999=: 'AAajodbuild999' [ coerase <'AAajodbuild999'
coinsert 'ijod'
```

```
NB. record open dictionaries and open JOD dictionaries
ooo=: }. did 0
od ;:'joddev jod utils' [ 3 od''
```

```
NB. get JOD build utilities and version tracking nouns
tmploc get }. grp 'buildjod'
```

```
NB. set distribution directories
jddir=: 'JODSTAGEDIR JODGITDIR JODSOURCESTAGEDIR JODSTAGEPDFDIR JODSTAGEDOCDIR'
jddir=: jddir , ' JODGITDOCDIR JODADDONDIR JODSCRIPTDIR JOEXTSDIR'
(jddir)=: setbuilddirs 0
```

```
NB. generate distribution scripts
updatejodmanifest 0
JODVMD buildjodcompressed JODSTAGEDIR;JODGITDIR;JODADDONDIR;JODSCRIPTDIR
JODTOOLSVM buildjodtoolscompressed JODSTAGEDIR;JOEXTSDIR;JODSCRIPTDIR
JODVMD updatejoddistribution JODSTAGEDIR;JODGITDIR
JODVMD updatejodsourcedumps JODSOURCESTAGEDIR
JODVMD releasejod JODSTAGEDIR;JODSTAGEPDFDIR;JODSTAGEDOCDIR;JODGITDOCDIR
```

```
NB. destroy build locale
cocurrent 'base'
coerase <tmploc_AAajodbuild999_
```



C JOD Classes

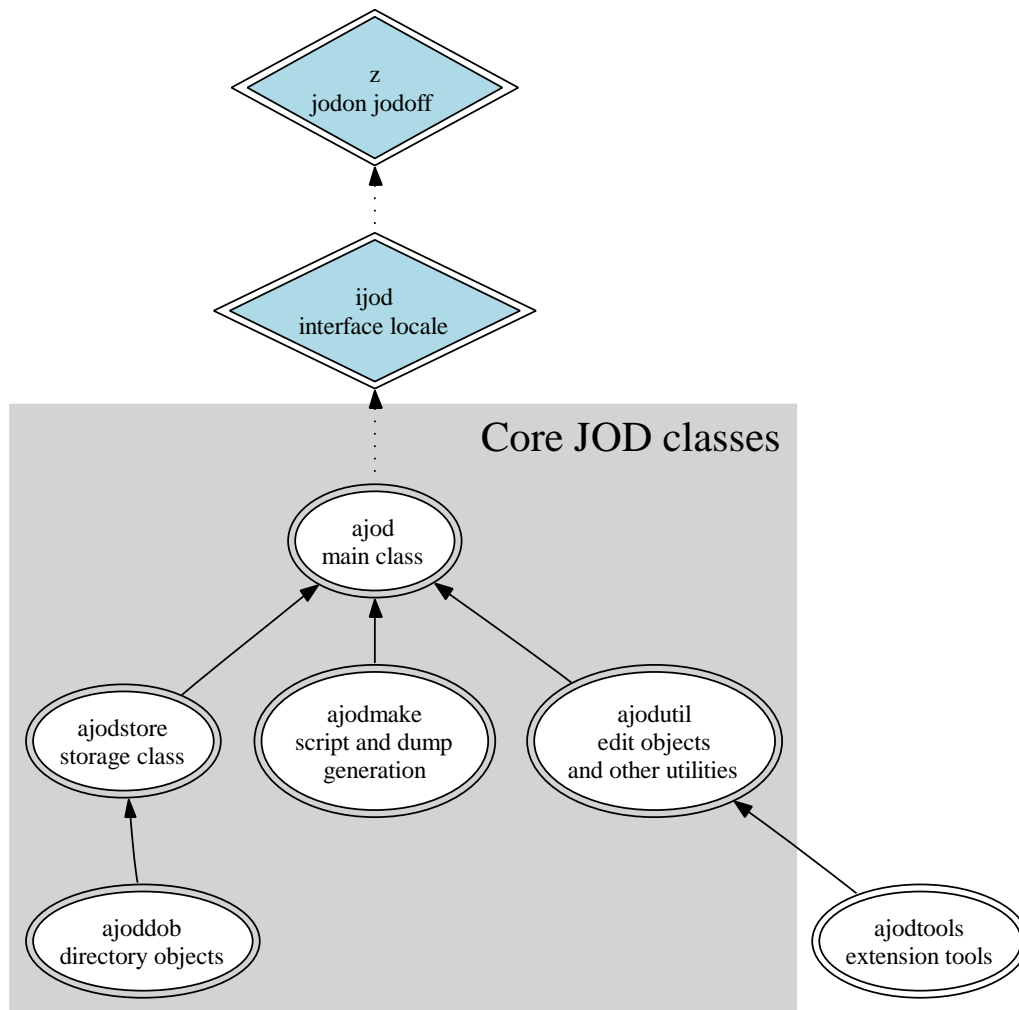


Figure 4: This diagram shows how JOD classes are related. JOD classes are loaded into J addon **a** locales. The arrows indicate how J names are resolved. The *blue diamond* locales `ijod` and `z` are on the base locale's path: `copath 'base'`. *Diagram Generated by Graphviz [9]*.

E 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 75, 75 and 76.

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	L ^A T _E X 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' NB. store html
XML	24	XML text code	4 put 'XMLtext24';'<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)
PYTHON	29	Python script text	4 put 'PYTHONtext';PYTHON_ajod_;'2 ** 1024'
SQL	30	SQL script text	4 put 'SQLtext';SQL_ajod_;'select * from yada'
JSON	31	JSON text	4 put 'JSONtext';JSON_ajod_;'{"json": "golden-fleece"}'

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*. For `get` and `put` a negative option code gets and puts timestamp arrays.

Negative Codes		
Code	Use	Example
<code>_1</code>	path order list	<code>0 _1 dnl ''</code> <i>NB. path order list of words</i>
<code>_2</code>	path order list	<code>1 _2 dnl 'boo'</code> <i>NB. path order list of test names containing boo</i>
<code>_14</code>	names and timestamps	<code>1 _14 get }. revo''</code> <i>NB. names and timestamps array</i>
<code>_14</code>	names and timestamps	<code>4 _14 put tsarray</code> <i>NB. update macro timestamps</i>

Table 12: JOD Negative Codes

F 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. The master file can be recreated with the call:
NB.
NB. createmast_ajod_ JMASTER_ajod_
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

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

NB. ROOTFOLDER is empty by default. If it is set to a (jpath) J configured

NB. folder ROOTFOLDER overrides default locations for (mls) generated scripts

ROOTFOLDER ; (character) redirects (mls) scripts to J folder ;

NB. typical nonempty setting

NB. ROOTFOLDER ; (character) redirects (mls) scripts to J folder ; ~user/jodroot

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. COPYRIGHT ; (character) ; All rights reserved

NB. MYPARAMETER ; (+integer) the answer ; 42

)



G 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.
NB. Note to J developers. A shutdown sentence (a line of J the
NB. interpreter executes before terminating) would be very
NB. useful.
NB.
NB. author: John D. Baker
NB. email: bakerjd99@gmail.com

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

NB. minimum print precision to show yyyymmdd dates (see jodage)
9!:11 [ 8

NB. set jqt windows console size - automatic for linux/mac/ios
Cwh_j_=: 140 24

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

NB. JOD interface locale - (ijod) is a good place for ad hoc JOD addons
coclass 'ijod'

NB. (ijod) error/ message text
ERRIJOD00=: 'current group name (jodg_ijod_) not set'
ERRIJOD01=: 'current suite name (jods_ijod_) not set'
OKIJOD00=: 'no matches'
```



NB. add delete from current group or current suite

NB. requires direct definition introduced with J 9.02

```
ag=: {{if. wex_ajod_ <'jodg' do. jodg addgrp y else. jderr_ajod_ ERRIJOD00 end.}}
as=: {{if. wex_ajod_ <'jods' do. (jods;3) addgrp y else. jderr_ajod_ ERRIJOD01 end.}}
dg=: {{if. wex_ajod_ <'jodg' do. jodg delgrp y else. jderr_ajod_ ERRIJOD00 end.}}
ds=: {{if. wex_ajod_ <'jods' do. (jods;3) delgrp y else. jderr_ajod_ ERRIJOD01 end.}}
```

NB. referenced words not in current group

```
nx=: 3 : 0
if. -.wex_ajod_ <'jodg' do. jderr_ajod_ ERRIJOD00 return. end.
if. badrc_ajod_ gn=. grp jodg do. gn return. end.
(allrefs }. gn) -. gn
)
```

NB. words in current group using a word

```
ug=: 3 : 0
if. -.wex_ajod_ <'jodg' do. jderr_ajod_ ERRIJOD00 return. end.
if. badrc_ajod_ gn=. grp jodg do. gn return. end.
y usedby }. gn
)
```

NB. generate current group and save load script

```
sg=: {{if. wex_ajod_ <'jodg' do. mls jodg else. jderr_ajod_ ERRIJOD01 end.}}
```

NB. top (put dictionary) words, groups in revision order

```
tw=: revo
tg=: 2&revo
```

NB. run tautology as plaintest - does not stop on nonzero results

```
rt=: 2&rtt
```

NB. run macro silently - will show explicit smoutput

```
rs=: 1&rm
```

NB. short help for group words

```
hg=: [: hlpnl [: }. grp
```

NB. short help on put objects in revised order from code:

NB. hr 4 NB. macro

NB. hr 2 NB. groups

NB. 10 hr 0 NB. last ten words

```
hr=: 3 : 0
```

```
if. badrc_ajod_ w=. y revo '' do. w return. end.
```

```
y hlpnl }. w
```

```
:
```

```
if. badrc_ajod_ w=. y revo '' do. w return. end.
```

```
y hlpnl x { . }. w
```



```
)

NB. search short help for string and list matching words
NB.      hss 'see long'    NB. search word short text
NB.  2   hss 'see long'    NB. search group short text
NB.  4   hss 'post'        NB. search macro short text
hss=: 3 : 0
0 hss y
:
if. badrc_ajod_ w=. x dnl '' do. w return. end.
d=. x hlpnl }. w
if. 0=#w=. 1 >@{ d do. ok_ajod_ OKIJOD00 return. end.
if. 0=#s=. I. (,:y) +./"1@E. w do. ok_ajod_ OKIJOD00 return. end.
s&{ &.> d
)

NB. single line explanation
NB.      slex 'word'        NB. word
NB.  4   slex 'jodregister' NB. macro
NB.  1   slex 'thistest'    NB. test
slex=: 3 : 0
0 slex y
:
if. badcl_ajod_ sl=. x disp y do. sl return. end.
(x,8) put y;firstcomment__JODtools sl
)

NB. regenerate put dictionary word cross references
reref=: 3 : 0
if. badrc_ajod_ n=. revo '' do. n return. end.
(n,.s) #~ -.;0{"1 s=.0 globs&> n=.}.n
)

NB. handy cl doc helpers
docscr=: {{ctl_ajod_ (61;0;0;'NB.') docct2__UT__JODobj };._1 LF,y-.CR}}
doctxt=: {{ctl_ajod_ (61;0;0;'') docct2__UT__JODobj };._1 LF,y-.CR}}

NB. display noun on screen and return noun value
showpass=:] [ 1!:2&2

NB. edit command
DOCUMENTCOMMAND=: 'showpass pr ''{~N~}'''

NB. read & write files
read=:1!:1&({\`<@.(32&>@{3!:0}))
write=:1!:2 ]\`<@.(32&>@{3!:0}))
readnoun=:3!:2@{1!:1&({\`<@.(32&>@{3!:0}))}
writenoun=:([ 3!:1 [) (1!:2 ]\`<@.(32&>@{3!:0})) ]
```



NB. fetch edit text/macros

```
tt=:] ; gt
mt=:] ; 25"_ ; gt    NB. *.txt
mj=:] ; 21"_ ; gt    NB. *.ijs
md=:] ; 27"_ ; gt    NB. *.markdown
mx=:] ; 22"_ ; gt    NB. *.tex
```

NB. examples of JOD session start ups - shows

NB. how to open dictionaries and invoke project macros

NB. set up current project (1 suppress IO, 0 or elided display)

NB. 1 rm 'prjsmughacking' [smoutput od ;:'smugdev smug utils'

NB. 1 rm 'prjjod' [smoutput od ;:'joddev jod utils'

cocurrent 'base'

coinset 'ijod'



H 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 words. 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.
```

```
smoutput 'joduserconfig.ijs executing ...'
```

```
NB. PDF reader in jodutil class - Reset for other PDF readers
```

```
PDFREADER_ajodutil_='C:\Adobe\Acrobat Reader DC\Reader\AcroRd32.exe'
```

```
NB. Reset J's PDF reader to match JOD's PDF reader - do for (jconsole.exe)
```

```
PDFReader_j_= PDFREADER_ajodutil_
```

```
NB. Alternative Ghostscript compatible reader
```

```
NB. PDFREADER_ajodutil_='C:\Program Files\Ghostgum\gsview\gsview32.exe'
```

```
NB. Preferred web browser in jodutil class - default Windows Firefox directory
```

```
WWW0_ajodutil_='C:\Program Files\Mozilla Firefox\firefox.exe'
```

```
NB. Secondary web browser in jodutil class - default Windows directory
```

```
WWW1_ajodutil_='C:\Program Files\Internet Explorer\IEXPLORE.EXE'
```

```
NB. Text editor to use when running JOD in (jconsole.exe) on Windows systems
```

```
NB. QT/JHS configurations are not necessarily applied for (jconsole.exe)
```

```
EDCONSOLE_ajodutil_='C:\Program Files (x86)\notepad++\notepad++.exe''
```



I 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 [44](#).

NB. WARNING: JOD managed section do not edit!

NB.<JOD_Load_Scripts>

```
buildpublic_j_ 0 : 0
bstats c:/jod/jutils/script/bstats.ijs
xmlutils c:/jod/utlils/script/xmlutils.ijs
analystgraphs c:/jod/franklin/script/analystgraphs.ijs
TeXfrWpxml c:/jod/docs/script/TeXfrWpxml.ijs
jodtester c:/jod/joddev/script/jodtester.ijs
waypoints c:/jod/gps/script/waypoints.ijs
Weeks c:/jod/docs/script/Weeks.ijs
MirrorXref c:/jod/smugpyter/script/MirrorXref.ijs
DudTeXPreprocess c:/jod/docs/script/DudTeXPreprocess.ijs
BiblioHelper c:/jod/docs/script/BiblioHelper.ijs
RecodeSchedZ c:/jod/mwecc/script/RecodeSchedZ.ijs
)
```

NB.</JOD_Load_Scripts>



J Turning JOD Dump Script Tricks

Dump script generation is my favorite JOD feature. Dump scripts serialize JOD dictionaries; they are mainly used to back up dictionaries and interact with version control systems: see appendix L on page 91. However, dump scripts are general J scripts and can do much more! Maintaining a stable of healthy JOD dictionaries is easier if you can turn a few dump script tricks.¹⁷ The following examples assume a configured folder JODTEST: see Section 2 on 5.

1. **Flattening reference paths:** Open JOD dictionaries define a reference path: see appendix D on page 74. For example, if you open the following dictionaries:

```
NB. open four dictionaries
od ;:'smugdev smug image utils'
+-----+
|1|opened (ro/ro/ro/ro) ->|smugdev|smug|image|utils|
+-----+
```

The reference path is /smugdev/smug/image/utils.

When objects are retrieved each dictionary on the path is searched in reference path order. If there are *no compelling reasons* to maintain separate dictionaries you can improve JOD retrieval performance and simplify dictionary maintenance by flattening all or part of the path.

To flatten the reference path do:

```
NB. reopen the first three dictionaries on the path
od ;:'smugdev smug image' [ 3 od ''
+-----+
|1|opened (ro/ro/ro) ->|smugdev|smug|image|
+-----+

NB. dump to a temporary file (df)
df=: {: showpass make jpath '~JODTEST/smugflat.ijs'
+-----+
|1|object(s) on path dumped ->|c:/jodtest/test/smugflat.ijs|
+-----+

NB. create a new flat dictionary
newd 'smugflat';jpath '~JODTEST/smugflat' [ 3 od ''
+-----+
|1|dictionary created ->|smugflat|c:/jodtest/test/smugflat/|
+-----+
```

¹⁷Spicing up one's rhetoric with a double entendre like "turning tricks" might be construed as a *microaggression*. The point of colorful language is to memorably make a point. You are unlikely to forget *turning dump script tricks*.

```

NB. open the flat dictionary and (utils)
od ;:'smugflat utils'
+-----+
|1|opened (rw/ro) ->|smugflat|utils|
+-----+

NB. reload dump script ... output not shown ...
0!:0 df

```

The collapsed path /smugflat/utils will return the same objects as the longer path. It is important to understand that the collapsed dictionary smugflat does not necessarily contain the same objects found in the three original dictionaries smugdev, smug and image. If objects with the same name exist in the original dictionaries only the first one found will be in the collapsed dictionary.

2. **Merging dictionaries:** If two dictionaries *contain no overlapping objects* it might make sense to merge them. This is easily achieved with dump scripts. To merge two or more dictionaries do:

```

NB. open and dump first dictionary
od 'dict0' [ 3 od ''
+-----+
|1|opened (rw) ->|dict0|
+-----+
df0=: {: showpass make jpath '~JODTEST/dict0.ijs'
+-----+
|1|object(s) on path dumped ->|c:/jodtest/test/dict0.ijs|
+-----+

NB. open and dump second dictionary
od 'dict1' [ 3 od ''
+-----+
|1|opened (rw) ->|dict1|
+-----+
df1=: {: showpass make jpath '~JODTEST/dict1.ijs'
+-----+
|1|object(s) on path dumped ->|c:/jodtest/test/dict1.ijs|
+-----+

NB. create new merge dictionary
newd 'mergedict';jpath '~JODTEST/mergedict' [ 3 od ''
+-----+
|1|dictionary created ->|mergedict|c:/jodtest/test/mergedict/|
+-----+

```



```
NB. open merge dictionary and run dump scripts
od 'mergedict'
++-----+
|1|opened (rw) ->|mergedict|
++-----+

NB. reload dump scripts ... output not shown ...
0!:0 df0
0!:0 df1
```

Be careful when merging dictionaries. If there are common objects the last object loaded is the one retained in the merged dictionary.

3. **Updating master file parameters:** When a new parameter is added to `jodparms.ijs`, see appendix F on page 77, it will not be available in existing dictionaries. With dump scripts you can rebuild existing dictionaries and update parameters. To rebuild a dictionary with new or custom parameters do:

```
NB. save current dictionary registrations
(toHOST ; 1 { 5 od '' ) write_ajod_ jpath '~JODTEST/jodregister.ijs'

NB. open dictionary requiring parameter update
od 'dict0' [ 3 od ''
++-----+
|1|opened (rw) ->|dict0|
++-----+

NB. dump dictionary and close
df=: {: showpass make jpath '~JODTEST/dict0.ijs'
++-----+
|1|object(s) on path dumped ->|c:/jodtest/test/dict0.ijs|
++-----+

3 od ''
++-----+
|1|closed ->|dict0|
++-----+

NB. erase master file and JOD object id file
ferase jpath '~addons/general/jod/jmaster.ijf'
1
ferase jpath '~addons/general/jod/jod.ijn'
1
```



```
NB. recycle JOD - this recreates (jmaster.ijf) and (jod.ijn)
NB. using the new dictionary parameters defined in (jodparms.ijs)
(jodon , jodoff) 1
1 1

NB. re-register dictionaries
load jpath '~JODTEST/jodregister.ijs'

NB. create a new dictionary - it will have the new parameters
newd 'dict0new'

NB. open the new dictionary - taking care to include all
NB. dictionaries on (dict0)'s reference path
od ;:'dict0new utils' [ 3 od ''

NB. reload dump script ... output not shown ...
0!:0 df
```

Before executing complex dump script procedures *back up your JOD dictionary folders* and play with dump scripts on test dictionaries. Dump scripts are essential JOD dictionary maintenance tools but like all powerful tools they must be used with care.



K JOD Direct Definition Support

J version 9.02 (February 2021) introduced *direct definitions*. The designers of APL languages, of which J is a member, introduced direct definitions long after *canonical* or notorious “∇ editor” style definitions. In retrospect, most agree the entire family of languages would be better off if direct definition had come first. Direct definitions are more elegant, concise, versatile, and beautiful than clunky canonical equivalents. They’re also easier to comprehend and compose than long J tacit definitions. But history is history, and software history is hard to ignore because of the *installed base*¹⁸. We’re stuck with our kludges!

In JOD’s case this shows up in how [globs](#), see page 36, classifies global and local names in words that contain direct definitions. Consider the following:

```
make_my_def=: 3 : 0

NB. local to make_my_def
here=. we=. 2 + are=. 3 * local=. 4

global_adv=: {{*./"1 u/&> 2 <\ "1 y}}

local_verb=. {{}}d
  NB. not local to make_my_def
  we=. are=. not=. make_my_def=. local=. x
  if. 1=:y do.
    NB. deep global to make_my_def
    {{deep_gbl=: 'deep';":y}} y
  end.
}}

0 local_verb y
)
```

`make_my_def` contains local and global direct definitions. When executed it creates `global_adv` and runs `local_verb` which in turn creates `deep_gbl`. When `local_verb` runs it creates a local namespace like any explicit J verb. Hence `not` and `make_my_def` are not *strictly* `make_my_def` locals. *The execution of directly defined local words is equivalent to calling explicit words.* `globs` does not track direct definition name scopes. It views all direct definition names as if they belonged to the topmost word. For `make_my_def` `globs` this gives:

```
NB. JOD name classification
11 globs 'make_my_def'
```

¹⁸Also known as *users*

```

+--+-----+
|1|+-----+-----+
| ||Global|+-----+-----+
| ||      ||deep_gbl|global_adv|
| ||      |+-----+-----+
| |+-----+-----+
| ||Local |+---+---+---+---+---+---+---+---+
| ||      ||are|here|local|local_verb|make_my_def|not|we||
| ||      |+---+---+---+---+---+---+---+---+
| |+-----+-----+
| ||(*)=: |
| |+-----+-----+
| ||(*)=. |
| |+-----+-----+
| ||for. |
| |+-----+-----+
+--+-----+

```

```

    NB. execute and show name classes
    make_my_def 1
+-----+
|deep|1|
+-----+
    nc ;:'local_verb global_adv deep_gbl'
_1 1 0

```

When embedding direct definitions in explicit words, or within other direct definitions, it's a good idea to make names distinct. For example, in embedded `local_verb` do something like:

```

local_verb=. {{}}d
    weLv=. areLv=. notLv=. make_my_defLv=. localLv=. x
    if. 1=:y do.
        NB. deep global to make_my_def
        {{deep_gbl=: 'deep';":y}} y
    end.
}}

```



L JOD and Version Control Systems

Despite JOD's backup and restore facilities, see `bnl`, `bget`, `packd` and `restd` on pages 19, 17, 52 and 56, JOD is not a source code version control system like `Git` [13] or `Fossil` [12]. 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 43, can dump entire dictionaries as a single J script. Dump scripts contain all¹⁹ 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.²⁰

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

1 rm 'dumpput' NB. run dump macro - (utils) must be on path
+-----+-----+
|l|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
```

¹⁹Word references are not present in dump scripts. They can be easily regenerated with `globs`, see page 36.

²⁰A 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 66. In the following a Git repository has been created in the script subdirectory and the contents of the `exim` 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 (exim) content and save changes ...

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

```



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

bakerjd99@NINJA /c/jod/smugdev/script (master)
$ git commit -m '(masspixels) added to description of (exim) interface'
[master e93ffe5] (masspixels) added to description of (exim) 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`²¹ writes individual JOD put dictionary word files. `pwf` is stored in the `utils` dictionary.

```
pwf=:3 : 0
```

```
NB.*pwf v-- write path dictionary words as script files.
NB.
NB. monad: pwf clPattern
NB.
NB. pwf '' NB. write all path 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_ mkdir_ajod_ write_ajod_
pk=. >@{
tsl=. ] , ('\'_' = {:) }. '\'_
if. badrc_ajod_ ws=. 0 _1 dnl y do. ws return. end.
if. badrc_ajod_ ws=. 0 10 get 1 pk ws do. ws return. end.
NB. individual word scripts using short description text for tacits
if. badrc_ajod_ ws=. 0 0 1 wtext__MK__JODobj 1 pk ws do. ws return. end.
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
mkdir_ajod_ <jpathsep_ajod_ tsl x=. ALI__DL, 'words'
end.
```

²¹`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** (91) level method.



```

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.
)

```

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(-)

```



M Hungarian Notation for J



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

M.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** [17]. 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.

M.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.²² Adverbs and conjunctions roughly correspond to the higher order functions

²²Adverbs and conjunctions also take noun arguments.

found in languages like **LISP** [11] and **Scheme** [15]. J *explicit* definition syntax reserves the characters `x y m n u v` for arguments: see Table 13 on page 96.²³ The Hungarian notation described here focuses on noun arguments, (`x` and `y`), because they are the most common.

J Explicit Arguments	
<code>x</code>	<i>left verb noun argument</i>
<code>y</code>	<i>right verb noun argument</i>
<code>m</code>	<i>left conjunction noun argument</i>
<code>n</code>	<i>right conjunction noun argument</i>
<code>u</code>	<i>left adverb/conjunction verb argument</i>
<code>v</code>	<i>right conjunction verb argument</i>

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 262144
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/unicode4'
(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:

²³Earlier 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*.²⁴
- 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'

M.3 Hungarian Noun Descriptions

To describe J nouns I use the following rules:²⁵

1. For basic descriptions I use *TypeRank[Name]* where *Type* comes from Figure 6 on page 98, *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 98 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
```

²⁴Rank 0, or 0-dimensional objects occur 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.

²⁵As in the film *Pirates of the Caribbean* these rules are more like *guidelines*!



J Native Type Prefixes		
Prefix	Native Type	(3! : 0) code
p	<i>boolean</i>	1
c	<i>literal</i>	2
i	<i>integer</i>	4
f	<i>floating</i>	8
j	<i>complex</i>	16
b	<i>boxed</i>	32
X	<i>extended</i>	64
R	<i>rational</i>	128
SP	<i>sparse boolean</i>	1024
SC	<i>sparse literal</i>	2048
SI	<i>sparse integer</i>	4096
SF	<i>sparse floating</i>	8192
SJ	<i>sparse complex</i>	16384
SB	<i>sparse boxed</i>	32768
s	<i>symbol</i>	65536
w	<i>unicode</i>	131072
W	<i>unicode4</i>	262144

(a) J native type prefixes

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

(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.

<code>blcl</code>	<i>boxed list of character lists</i>
<code>blit</code>	<i>boxed list of integer tables</i>
<code>bljtCoord</code>	<i>boxed list of complex tables</i>
<code>blul</code>	<i>boxed list any lists</i>
<code>b[3]s[4]Maps</code>	<i>boxed rank 3 array of rank 4 symbol array Maps</i>

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.

<code>(<blcl), <jtPlane</code>	<i>two item list</i>
<code>pa;ftXy;<btuu</code>	<i>three item list</i>
<code>cl;ia;(<blcl), <blcl</code>	<i>see Table 9 page 70</i>
<code>itYYYYMMDD;slWords;(<bt), <clName</code>	<i>four item list</i>
<code>saRed, saGreen, saBlue</code>	<i>emphasize items of simple noun</i>

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.²⁶

```

dyadic put argument description see page 52
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 27
iaObject dnl z1 ✓ clPstr
(iaObject, iaOption) dnl z1 ✓ clPstr
(iaObject, iaOption, iaQualifier) dnl z1
(iaObject, iaOption, iaQualifier) dnl clPstr

```

²⁶The `✓` symbol was chosen because it falls outside of J's ASCII vocabulary and suggests "either-or."

N JOD Mnemonics

“Mnemonics Neatly Eliminate Man’s Only Nemesis - Insufficient Cerebral Storage.”

jodhelp us!
I get it!
dnl is not just a river in Egypt.
And Jesse bget old code.
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.
bget me a backup shrubbery.



References

- [1] Chris Burke. Pacman J Package Manager, 2019.
J Wiki documentation about Pacman: the J package manager. Early versions of Pacman were called JAL but Pacman is the preferred term. Pacman is the main tool for installing and maintaining J addons.
<https://code.jsoftware.com/wiki/Pacman>.
- [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.
Warning: this URL is not private.
<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 (2020) 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.
<https://www.jsoftware.com/help/dictionary/title.htm>.
- [4] John D. Baker. jod addon, September 2008.
J Wiki download page for the jod addon.
<https://www.jsoftware.com/jwiki/Addons/general/jod>.
- [5] John D. Baker. jodsource addon, September 2008.
J Wiki download page for the jodsource addon.
<https://www.jsoftware.com/jwiki/Addons/general/jodsource>.
- [6] John D. Baker. joddocument addon, June 2012.
J Wiki download page for the joddocument addon.
<https://www.jsoftware.com/jwiki/Addons/general/joddocument>.
- [7] John D. Baker. *The JOD Page*, March 2020.
 This website maintains references to all JOD related documents and downloads.
<https://analyzethedatanotthedrive1.org/the-jod-page/>.
- [8] Donald Knuth. Personal website, 2008.
 The website of the legendary computer scientist Donald Knuth. Knuth created the typesetting language T_EX in the 1970's and T_EX is still going strong! Genius is hard to replace!
<https://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!
<https://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, also called Pacman, is the main tool for downloading and installing J addons.
https://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.
<https://www.cs.cmu.edu/Groups/AI/html/cltl1/cltl2.html>.
- [12] Fossil Website. Official Fossil website, 2019.
The Fossil version control website.
<https://www.fossil-scm.org/home/doc/trunk/www/index.wiki>.
- [13] Git Website. Official Git website, 2012.
The Git version control website.
<https://git-scm.com/>.
- [14] Jupyter Website. Official Jupyter website, 2019.
Jupyter notebooks are open source *literate* programming and documentation tools. They have attracted a wide following and are one of the more significant software developments in decades.
<https://jupyter.org/>.
- [15] Scheme Website. Scheme, 2019.
All about the *Scheme* programming language.
<https://www.scheme.com/tspl4/>.
- [16] J Software Wiki. *NuVoc Wiki*, March 2020.
The *NuVoc J Wiki* is the single best current “document” describing the J programming Language. It is *zealously* maintained and refined by the J community.
<https://code.jsoftware.com/wiki/NuVoc>.
- [17] Wikipedians. Hungarian notation, 2008.
Wikipedia entry for *Hungarian Notation*: it's overwrought but conveys the essentials.
https://en.wikipedia.org/wiki/Hungarian_notation.



List of Tables

1	Abbreviated Document Version History	1
2	JOD generated script structure	61
3	jmaster.ijf file component layout	63
4	jwords.ijf file component layout	64
5	jtests.ijf file component layout	65
6	jgroups.ijf file component layout	67
7	jsuites.ijf file component layout	68
8	jmacros.ijf file component layout	69
9	juses.ijf file component layout	70
10	JOD Object Codes	75
11	JOD Qualifier Codes	75
12	JOD Negative Codes	76
13	J Arguments	96

List of Figures

1	JOD Folders	7
2	JOD Labs	8
3	JOD Directories	66
4	JOD Classes	73
5	Zippy the Pinhead	95
6	Hungarian Type Prefixes	98



Index

- Addon, [4](#), [6](#)
- assignments
 - indirect, [21](#)
- backup
 - abv, [16](#)
 - bget, [17](#)
 - bnl, [19](#)
 - packd, [52](#)
 - restd, [56](#)
- binary, *see* jfiles
- building
 - JOD distribution, [72](#)
- classes, [73](#)
- codes
 - argument, [75](#)
- comment tag
 - (*) = ., [21](#)
 - (*) = :, [21](#)
 - (-.) = :, [22](#)
 - (/:) = :, [22](#)
 - (<:) = :, [37](#)
- compression, [20](#)
- configuration
 - jodprofile.ijs, [79](#)
 - joduserconfigbak.ijs, [83](#)
 - startup.ijs, [84](#)
 - J folders, [6](#), [7](#)
 - tool, [6](#)
- cross references
 - getrx, [36](#)
 - globs, [36](#)
 - uses, [59](#)
 - names, [21](#)
- dates, [41](#)
- dependent section, [33](#), [61](#)
- dictionary
 - create
 - newd, [47](#)
 - dump
 - make, [43](#)
 - open
 - od, [50](#)
 - parameters, [30](#), [87](#)
 - search
 - bnl, [19](#)
 - dnl, [27](#)
 - mn1, [45](#)
 - source
 - jodsource, [71](#), [72](#)
- DIDNUM, [63](#), [70](#), [74](#)
- direct definition, [89](#)
- documentation
 - doc, [28](#)
 - hlpnl, [39](#)
- dump scripts, [60](#), [85](#)
- edit
 - ed, [32](#)
 - et, [33](#)
 - gt, [39](#)
 - nt, [49](#)
 - nw, [48](#)
- group
 - addgrp, [17](#)
 - delgrp, [25](#)
 - grp, [38](#)
 - lg, [42](#)
 - locgrp, [42](#)
- help
 - hlpnl, [39](#)
 - jodhelp, [41](#)
- ijod interface, [16](#), [73](#)
- installation

- The JOD Page*, 5
- distribution, 71
- pacman, 5
- jfiles
 - jgroups.ijf, 67
 - jmacros.ijf, 69
 - jmaster.ijf, 63
 - jsuites.ijf, 68
 - jtests.ijf, 65
 - juses.ijf, 70
 - jwords.ijf, 64
- labs, 6, 11
- load scripts, 44
- locale
 - base, 57
 - classes, 73
 - references, 59
- macros
 - jmacros.ijf, 69
 - rm, 57
- master file, 30, 52, 62, 77, 87
- parameters, *see* dictionary
 - dpset, 30
- path order list, 27, 28, 76
- postprocessor, 42, 44, 61, 75
- put dictionary, 10, 12, 17, 24, 37, 52, 71, 74
- reference path, 38, 43, 70, 74, 85
- registration
 - regd, 55
- regular expressions
 - rxs, 58
- scope tags, 21
- scriptdoc, 28
- scripts
 - dump
 - make, 43
 - generation
 - lg, 42
 - make, 43
 - mls, 44
 - structure, 61
 - headers
 - put, 53
 - search
 - groups
 - locgrp, 42
 - names
 - bnl, 19
 - dnl, 27
 - mn1, 45
 - text
 - rxs, 58
 - source code, 71
 - tautology, 57
 - test suite, 4, 32, 39, 57
 - testing
 - get, 35
 - grp, 38
 - put, 52
 - rtt, 57
 - uses
 - references, 59
 - uses union
 - uses, 59
 - version control
 - Fossil, 91
 - Git, 91
 - z interface, 73
 - Zippy, 95

