Variable Verbosity Printing: messages.lisp

Roy M. Turner

Spring, 2021

Description

This file defines macros, a class, and methods for printing at various levels of verbosity. By using different macros for different verbosity levels, and by setting the global verbosity, you can control what is printed from your code for various purposes.

For example, though I often disparage the use of "debugging by print statement" in general (when you have access, like you do in Lisp, to a good debugger), it is often still really good to be able to quickly see what your code is doing when it's misbehaving *without* having set breakpoints to drop into the debugger or to use single-stepping and tracing. For this reason I often insert debugging statements in my code to let me know what it is doing, the value of important variables, etc. Rather than commenting these out when they are (perhaps temporarily) not needed, I can just change the verbosity from "debugging" to "normal".

All messages from the statements are ultimately produced by methods of an instance of the class `message-handler`; by default, the instance used is the one created by this file and stored in the global (dynamic) variable `*message-handler*`. A message handler tracks the current verbosity level, indentation (which you can change as well), and destination for the messages (the stream pointed to by the Lisp-defined `*standard-output*` variable by default). Macros are provided here to not only make it easy to use message handling, but also to allow there to be no method calls when the verbosity is lower than the message's.

Since `*message-handler*` is a dynamic variable, you are free to rebind it, e.g., using `let,` at any time to a different instance of `message-handler` with (e.g.) a different destination for messages. So if you have a function such as:

```
(defun foo (file)
  (with-open-file (out file :direction :output :if-does-not-exist :create
   :if-exists :supersede)
    (let ((*message-handler* (make-instance 'message-handler
       :destination out)))
      (msg out "Hi there!"))))
```

the string "Hi there" will be sent to the file when called, and when the function exits, messages will go back to being directed to wherever the default message handler is set to send them.

Message macros

The message macros are of two kinds: ones that take zero or more arguments and print each one separately (unformatted message macros); and ones (formatted message macros) that take at least one argument, a format string of the type used by the `format` function, with additional arguments used also like `format`, i.e., to provide data to use in the format string.

The unformatted message macros are:

- `msg`: prints when verbosity is at least `:normal` (see below for how to set the verbosity)
- `vmsg`: prints when verbosity is at least `:verbose`
- `dmsg`: prints when verbosity is at least `:debugging`
- `vdmsg`: prints when verbosity is `:verbose-debugging`

Any argument to the macro except the symbol `t` prints immediately following any previous arguments. The symbol `t` is used to specify a new line. Thus

```
(msg 'hi 'there 'bob)
```

would print:

```
HITHEREBOB
```

whereas

```
(msg 'hi " " 'there t 'bob)
```

would print

```
HI THERE
BOB
```

The formatted message macros are `fmsg`, `vfmsg`, `dfmsg`, and `vdfmsg`, with behavior corresponding to their unformatted counterparts with respect to verbosity. As an example,

```
(vfmsg "Hi there, ~a!" 'bob)
```

would print

```
Hi there, BOB!
```

By default, the formatted messages print on separate lines, with a line break, if needed, before printing and one afterward. If you prefer to control where all line breaks happen, you can change this behavior by using the `fmsg-inserts-line-breaks` macro with the argument `t`.

Creating message handlers

A message handler instance is created when you load this file and stored in the `*message-handler*` variable. To create a new message handler, instantiate `message-handler` with the parameters you want; you can let everything default by simply doing:

```
(setq *message-handler* (make-instance 'message-handler))
```

There are several keyword parameters that can be set when instantiating the message handler:

- `:destination` – Set this to a stream where you want messages to go. It defaults to `*standard-output*`.
- `:verbosity` – Set this to the verbosity you want, one of `:silent :normal :verbose :debugging`, or `:verbose-debugging`. By default, verbosity is `:normal`. If you set it to `:silent`, none of the message macros will print anything.
- `:fmsg-inserts-line-breaks` – Set this to `t` if you want a line break to be output after every formatted message macro, to `nil` if you do not. The default is `t`
- `:indentation` – How far messages should be indented from the left margin; controls how many spaces are output prior to messages. The default is 0.
- `:indentation-delta` – Set this to how many spaces you want each call to `with-indent` or `indent-messages` to increase indentation; default is 2.

Changing message handling behavior

This file provides some macros to change aspects of how messages are handled as well as to access some settings of the message handler instance. These are:

- `(set-destination` *stream* `)` – Set the output destination to *stream*.
- `(destination)` – Returns the current destination.
- `(silence-messages)`, `(silent-messages)`, and `(no-messages)` – These all do the same thing: turn off all messages.
- `(normal-messages)`, `(verbose-messages)`, `(debugging-messages)` – These set the verbosity correspondingly
- `(verbose-debugging-messages)` and `(all-messages)` – These both set the verbosity to print all kinds of messages.
- `(verbosity)` – Returns the current verbosity level.
- `(fmsg-inserts-line-breaks` *t/nil* `)` –
- `(set-indentation` *num* `)` – Sets the number of spaces to precede messages.

- `(set-indentation-delta` *num* `)` – Sets the number of spaces `indent` and `with-indentation` adds to the current indentation (and that `deindent` subtracts).
- `(with-indentation` *form** `)` – This is used to "wrap" the forms (i.e., Lisp "statements") in an indentation level. For example:

  ```
  (fmsg "hi")
  (with-indentation
      (fmsg "there")
      (with-indentation
          (fmsg "Bob")))
   (fmsg "how's it going?")
  ```

  would print:

  ```
  hi
    there
      Bob
  how's it going?
  ```

- `(indent)` – Indent future output by the current indentation + the indentation-delta spaces.
- `(deindent)` – Indent future output by the current indentation − the indentation-delta spaces.
- `(with-destination` *form** `)` – Change the output destination for any message macro called in the forms (or anything they call); see example above.

Loading and using the macros

As with the `new-symbol.lisp` file, this file defines a new package, `message`, in which all macros, the message handler class, and methods are defined. To load the file:

```
(load "message")
```

Unless you import the macros, etc., you want from the message package, you will need to prefix them with the package name or nickname (`msg`), e.g.:

```
(msg:msg 'hi)
(message:fmsg "there")
```

You can import the symbols you want to use with the `import` function, e.g.,

```
(import '(msg:msg msg:fmsg))
```

or you can import all exported symbols (i.e., the ones you want) with:

```
(use-package 'message)
```

Code

Set up the package for the messages:

```
(unless (find-package "MSG")
  (defpackage "MESSAGE"
    (:use "COMMON-LISP")
    (:nicknames "MSG"))
    )

(in-package msg)

```

Here are all the macro definitions. Since they are used in the file, they need to come before their use (unlike functions, which can appear after their use in the code).[1] By the way, if you ever want to see what a macro call turns into, you can do:

[1] But not, of course, after they are actually *called*!

```
(macroexpand '(msg:msg t 'hi))
```

or similar.

```
(defmacro string-append (&rest l)
    `(concatenate 'string ,@l))

(defmacro no-messages? ()
  `(eql :silent (slot-value *message-handler* 'verbosity)))

(defmacro verbose? ()
  `(not (member (slot-value *message-handler* 'verbosity) '(:silent :normal))))

(defmacro silent? ()
  `(eq (slot-value *message-handler* 'verbosity) :silent))


(defmacro debugging? ()
  `(not (member (slot-value *message-handler* 'verbosity) '(:silent :normal :verbose))))

(defmacro verbose-debugging? ()
  `(eql (slot-value *message-handler* 'verbosity) :verbose-debugging))

(defmacro normal-messages ()
  `(setf (slot-value *message-handler* 'verbosity) :normal))

(defmacro silence-messages ()
  `(setf (slot-value *message-handler* 'verbosity) :silent))

(defmacro silent-messages ()
  `(setf (slot-value *message-handler* 'verbosity) :silent))

(defmacro no-messages ()
  `(setf (slot-value *message-handler* 'verbosity) :silent))

(defmacro verbose-messages ()
  `(setf (slot-value *message-handler* 'verbosity) :verbose))

(defmacro debugging-messages ()
  `(setf (slot-value *message-handler* 'verbosity) :debugging))

(defmacro all-messages ()
  `(setf (slot-value *message-handler* 'verbosity) :verbose-debugging))

(defmacro verbose-debugging-messages ()
  `(setf (slot-value *message-handler* 'verbosity) :verbose-debugging))

(defmacro msg (&rest l)
  `(unless (no-messages?)
     (unformatted-message *message-handler* ,@l)))

(defmacro vmsg (&rest l)
  `(when (verbose?)
     (unformatted-message *message-handler* ,@l)))

(defmacro dmsg (&rest l)
  `(when (debugging?)
     (unformatted-message *message-handler* ,@l)))

(defmacro vdmsg (&rest l)
  `(when (verbose-debugging?)
     (unformatted-message *message-handler* ,@l)))

(defmacro fmsg (string &rest l)
  `(unless (silent?)
     (formatted-message *message-handler* ,string ,@l)))

(defmacro vfmsg (string &rest l)
  `(when (verbose?)
     (formatted-message *message-handler* ,string ,@l)))

(defmacro dfmsg (string &rest l)
  `(when (debugging?)
     (formatted-message *message-handler* ,string ,@l)))

(defmacro vdfmsg (string &rest l)
  `(when (verbose-debugging?)
     (formatted-message *message-handler* ,string ,@l)))

(defmacro set-destination (stream)
  `(setf (slot-value *message-handler* 'destination) ,stream))

(defmacro destination ()
  `(slot-value *message-handler 'destination))

(defmacro verbosity ()
  `(slot-value *message-handler* 'verbosity))

(defmacro fmsg-inserts-line-breaks (&optional (value t))
  `(setf (slot-value *message-handler*) ,value))

(defmacro set-indentation (num)
  `(setf (slot-value *message-handler* 'indentation) ,num))

(defmacro set-indentation-delta (num)
  `(setf (slot-value *message-handler* 'indentation-delta) ,num))

```

The following is an example of how to “wrap” some code in some other code, like you see with `with-slots` and `with-open-file`. The trick is to put the code itself, prior to execution, inside an `unwind-protect` form. What that does is *always* execute its second argument no matter what—even if there are errors. To do that, you have to group the code you want to protect (thus the `progn`), *and* you don't want the code evaluated until after the `unwind-protect` has been started (thus it needing to be done in a macro).

```
(defmacro with-indentation (&rest l)
  `(progn
     (indent)
     (unwind-protect
       (progn ,@l)
     (deindent))))
```

```

(defmacro with-indent (&rest l)
  `(with-indentation ,@l))

(defmacro indent ()
  `(push-indentation *message-handler*))

(defmacro deindent ()
  `(pop-indentation *message-handler*))

(defmacro with-destination (dest &rest l)
  `(progn
     (push-destination *message-handler* ,dest)
     (unwind-protect
       (progn ,@l)
     (pop-destination *message-handler*))))
```

The message handler class. The two variables `indentation-stack` and `destination-stack` hold past indentations and destinations so they can be restored. These are used by the `with-xxx` macros above.

```
(defclass message-handler ()
  (
   (destination :initform *standard-output* :initarg :destination)
   (verbosity :initform :normal :initarg :verbosity)
   (fmsg-inserts-line-breaks :initform t :initarg :fmsg-inserts-line-breaks)
   (indentation :initform 0 :initarg :indentation)
   (indentation-delta :initform 2 :initarg :indentation-delta)
   (indentation-stack :initform nil)
   (destination-stack :initform nil)
   )
  )

```

These forms are used by the `with-xxx` macros to push and pop indentations and destinations.

```
(defmethod push-indentation ((self message-handler))
  (with-slots (indentation indentation-stack indentation-delta) self
    (push indentation indentation-stack)
    (setq indentation (+ indentation indentation-delta))))

(defmethod pop-indentation ((self message-handler))
  (with-slots (indentation indentation-stack) self
    (setq indentation (or (pop indentation-stack) 0))))

```

```
(defmethod push-destination ((self message-handler) dest)
  (with-slots (destination destination-stack) self
    (push destination destination-stack)
    (setq destination dest)))

(defmethod pop-destination ((self message-handler))
  (with-slots (destination destination-stack) self
    (setq destination (or (pop destination-stack) *standard-output*))))

```

This method uses `format` to send formatted messages to the message handler's destination.

```
(defmethod formatted-message ((self message-handler) format-string &rest args)
  (with-slots (destination) self
      (apply #'format
      (cons destination
    (cons (prepare-string self format-string) args)))))

```

This method prepares a string to be printed by inserting the correct number of spaces for the current indentation and by adding a newline on the end, if necessary. Note that I also have used the `~T` format and `~%` directives to do this.

```
(defmethod prepare-string ((self message-handler) string)
  (indent-string self (add-line-break-or-not self string)))

(defmethod indent-string ((self message-handler) string)
  (string-append (indentation-string self) string))

(defmethod indentation-string ((self message-handler))
  (with-slots (indentation) self
    (if (zerop indentation)
      ""
      (make-string indentation :initial-element #\Space))))

(defmethod add-line-break-or-not ((self message-handler) string)
  (with-slots (fmsg-inserts-line-breaks) self
    (if (not fmsg-inserts-line-breaks)
      string
      (string-append string (make-string 1 :initial-element #\Newline)))))

```

This method handles unformatted messages.

```
(defmethod unformatted-message ((self message-handler) &rest args)
  (with-slots (destination) self
    (dolist (arg (cons (indentation-string self) args))
      (if (eql 't arg)
 (fresh-line destination)
 (write arg :stream destination :escape nil)))))

```

These are the symbols that are exported, that is, that are external to this package and that thus can be imported (using `import`, e.g.) into your package:

```
(export '(msg
   dmsg
   vmsg
   vdmsg
   fmsg
   vfmsg
   dfmsg
   vdfmsg
   *message-handler*
   message-handler
   set-destination
   destination
   verbosity
   fmsg-inserts-line-breaks
   set-indentation
   set-indentation-delta
   with-indentation
   indent
   deindent
   with-destination
   normal-messages
   silence-messages
   silent-messages
   no-messages
   verbose-messages
   debugging-messages
   verbose-debugging-messages
   all-messages
   ))
```

Now, set up a message handler. Note that every time you reload this file, a new instance is created.

```
(defparameter *message-handler* (make-instance 'message-handler))
```