SLF4O

version 0.1.0-SNAPSHOT, April 2020

Andrew Janke

This manual is for SLF4O, version 0.1.0-SNAPSHOT.

Short Contents

Table of Contents

1 Setup

To use SLF4O in your code:

1. Install and load the SLF4O package using Octave's `pkg` command
   `pkg install https://github.com/apjanke/octave-slf4o/archive/master.zip`
2. Load the SLF4O package in your Octave session
   `pkg load slf4o`
3. Call `logger.initSLF4O` to initialize the library before doing any logging calls
4. Add calls to the `logger.*` functions in your code

2 API

SLF4O provides:

- A set of logging functions to log events at various levels. This includes Regular and `"j"` variants for sprintf-style or SLF4J-style formatting.
- A Logger class for doing logging with more control over its behavior.
- A Log4j configurator tool and GUI.
- `dispstr`, a customizable string-conversion API.

All the code is in the `+logger` package. I chose a short, readable package name because if you're using logging, it'll show up a lot in your code.

2.1 Logging functions

Each logging level has a corresponding `logger.*` function and "J Variant" function that you can call to emit a log message at that level.

Level	**Function**	**J Variant**
ERROR	`logger.error`	`logger.errorj`
WARNING	`logger.warn`	`logger.warnj`
INFO	`logger.info`	`logger.infoj`
DEBUG	`logger.debug`	`logger.debugj`
TRACE	`logger.trace`	`logger.tracej`

The logging levels are predefined. They cannot be customized. This is a design decision made to keep the API simple and performant.

2.1.1 Calling logging functions

In your code, put calls to `logger.info(...)`, `logger.debug(...)`, and so on, as appropriate.

```
...
logger.info('Working on item %d of %d: %s', i, n, description);
logger.debug('Intermediate value: %f', someDoubleValue);
...
```

2.1.2 Regular and j variants

The regular ("m") versions of the logging functions take `fprintf`-style formatting and arguments, with `%s`/`%f`/`%d`/etc placeholders. These calls look like normal Octave `fprintf()` calls. The argument conversion and formatting is done at the Octave level before the message is passed along to the SLF4J Java library. These are the functions you should usually be using.

There are also `"j"` variants (`"j"` is for "Java") of all the the logging functions which use SLF4J style formatting. These use `"{}"` as the placeholders, and the arguments are passed down to the SLF4J Java layer to be converted there. These variants are useful if you're working with actual Java objects in your Octave code, and you want Java to handle the type conversion. In the `"j"` variants, all the input arguments are converted to Java objects using Octave's default auto-conversion.

Some Octave objects may not convert to Java objects at all, so you'll get errors when trying to use the `"j"` variants with them.

```
>> d = database;
>> logger.infoj('My database: {}', d)
No method 'info' with matching signature found for class 'org.slf4j.impl.Log4jLoggerAd
Error in logger.Logger/infoj (line 146)
        this.jLogger.info(msg, varargin{:});
Error in loggerCallImpl (line 69)
                logger.infoj(msg, args{:});
Error in logger.infoj (line 13)
loggerCallImpl('info', msg, varargin, 'j');
```

To avoid this, use the regular variants.

In both cases, the formatting and conversion is done lazily: if the logger is not enabled at the level you are logging the event, the function returns without doing the conversion. So you only pay the cost of the `sprintf()` or Java conversion and formatting if the logger is enabled.

2.1.3 Logger names

The logging functions in `+logger` use the caller's class or function name as the logger name. (This is in line with the Java convention of using the fully-qualified class name as the logger name.) This is accomplished with a trick using `dbstack`, looking back up the call stack to see who invoked it.

You can use anything for a logger name; if no logger of that name exists, one is created automatically. Logger names are arranged in a hierarchy using dot-qualified prefixes, like package names in Java or Octave. For example, if you have the following loggers:

- `foo.Thing`
- `foo.bar.Thing`
- `foo.bar.OtherThing`
- `foo.bar.baz.Whatever`

Then:

- All these loggers are children of the logger `foo`
- `foo.bar.Thing` and `foo.bar.OtherThing` are children of `foo.bar`, which in turn is a child of `foo`.
- `foo.bar.baz.Whatever` is a child of `foo.bar.baz`, which is a child of `foo.bar`, which is a child of `foo`.

2.1.3.1 The Logger object

You can also use the object-oriented `logger.Logger` API directly. This allows you to set custom logger names. It'll also be a bit faster, because it doesn't have to spend time extracting the caller name from the call stack. To use the Logger object directly, get a logger object by calling `logger.Logger.getLogger(name)` where `name` is a string holding the name of the logger you want to use.

```
logger = logger.Logger.getLogger('foo.bar.baz.MyThing');
```

```
logger.info('Something happened');
```

If you use `logger.Logger` in object-oriented Octave code, I recommend you do it like this, which looks like the SLFJ Java conventions.

```
classdef CallingLoggerDirectlyExample

    properties (Constant, Access=private)
        log = logger.Logger.getLogger('foo.bar.baz.qux.MyLoggerID');
    end

    methods
        function hello(this)
            this.log.info('Hello, world!');
        end

        function doWork(this)
            label = 'thingy';
            x = 1 + 2;
            timestamp = datetime;
            this.log.debug('Calculation result: label=%s, x=%f at %s', label, x, times
         end
    end

end
```

Even though `log` is a `Constant` (static) property, I like to call it via `this` because it's more concise, and then you can copy and paste your code that makes logging calls between classes. Make the `log` property `private` so you can have `log` properties defined in your subclasses, too; they may want to use different IDs.

3 The dispstr API

In addition to the SLF4J adapter layer, SLF4O provides a new API for generic value formatting and customizing the display of user-defined objects. This consists of a pair of functions, `dispstr` and `dispstrs`. They take values of any type and convert them to either a single string, or an array of strings corresponding to the input array's elements.

This is the equivalent of Java's `toString()` method, which is defined for almost everything and customized extensively. (Well, really it's equivalent to Java's `""+x` string concatenation operation, which really is defined for everything.)

```
str = dispstr(x)     % Returns char string
strs = dispstrs(x)   % Returns cellstr array
```

The input `x` may be *any* type.

Normally when writing a library, I avoid defining any global functions, to avoid polluting the shared namespace. But `dispstr` and `dispstrs` *must* be global functions, because they are polymorphic over all input types, including those which are themselves unaware of `dispstr`.

This provides an extension point for defining custom string conversions for your own user-defined classes. You can override `dispstr` and `dispstrs` in your classes, and SLF4O will recognize it. I find this is useful for other string formatting, too.

For uniformity, if you define `dispstr` in a class, I recommend that you override `disp` to make use of it. And you'll typically want to make `dispstr` and `dispstrs` consistent.

```
function disp(this)
    disp(dispstr(this));
end

% Standard implementation of dispstr
function out = dispstr(this)
    if isscalar(this)
        strs = dispstrs(this);
        out = strs{1};
    else
        out = sprintf('%s %s', size2str(size(this)), class(this));
    end
end
```

As a convenience, there is a `logger.Displayable` mix-in class which takes care of this boilerplate for you. It provides standard implementations of `disp` and `dispstr` in terms of `dispstrs`. If you inherit from `logger.Displayable`, you only need to define `dispstrs`.

3.1 The dispstr interface

The `dispstr` function/method takes a single argument, which may be an array of any size, and returns a single one-line string.

The `dispstrs` function/method takes a single argument, which may be an array of any size, and returns a `cellstr` array of exactly the same size as the input. For `strs = dispstrs(x)`, the string in `strs{i}` corresponds to the input `x(i)`.

3.1.1 How dispstr and SLF4O interact

When you call the normal (`"m"`) variants of the logging functions, `dispstr()` is applied to any inputs which are objects, so they're converted automatically and may be passed as parameters for the `%s` conversion. (In the normal Octave `sprintf`, most objects cannot be passed to `%s`; it results in an error.)

```
d = database;
logger.info('Database: %s', d);
```

For most Octave-defined objects, this just results in a `"m-by-n <classname>"` output. (But at least it doesn't raise an error, which is especially problematic when your functions are receiving inputs of the wrong type.) It gets particularly useful when you define custom `dispstr` overrides so your objects have useful string representations.

4 Configuration

All the actual logging goes through the Log4j back end; you can configure it as with any Log4j installation. See the Log4j 1.2 documentation (`http://logging.apache.org/log4j/1.2/` for details. (Note: you have to use the old 1.2 series doco, because that's what SLF4O currently ships with, due to a desire to maintain parity with SLF4J, which is constrained by the Log4j version that Matlab ships with.)

The `logger.Log4jConfigurator` class provides a convenient Octave-friendly interface for configuring Log4j to do basic stuff. It's enough for simple cases. But all the configuration state is passed on the the Log4j back end; none of it is stored in the Octave layer.

5 Implementation notes

I chose Log4j as the back end because that's what SLF4M uses, and I wanted to be maximally compatible with SLF4M. SLF4M chose Log4j as the back end because that's what ships with Matlab.

Aside from the `dispstr` formatting, everything is done purely in terms of the underlying SLF4J interface, so SLF4O is compatible with any other code or tools that use SLF4J or Log4j.

6 API Reference

6.1 API by Category

6.1.1 Logging

6.1.2 Dispstr

6.1.3 Uncategorized

6.2 API Alphabetically

6.2.1 dispstr

`out = dispstr (x)` [Function]
`out = dispstr (x, options)` [Function]

This returns a one-line string representing the input value, in a format suitable for inclusion into multi-element output. The output describes the entire input array in a single string (as opposed to dumping all its elements.)

The intention is for user-defined classes to override this method, providing customized display of their values.

The input x may be a value of any type. The main DISPSTR implementation has support for Matlab built-ins and common types. Other user-defined objects are displayed in a generic "m-by-n <class> array" format.

Options may be a struct or an n-by-2 cell array of name/value pairs (names in column 1; values in column 2).

Returns a single string as char.

Options: QuoteStrings - Put scalar strings in quotes.

Examples:

```
dispstr(magic(3))
```

See also: DISPSTRS, SPRINTFDS

6.2.2 dispstrs

`out = dispstrs (x)` [Function]
`out = dispstrs (x, options)` [Function]

Get display strings for array elements.

DISPSTRS returns a cellstr array containing display strings that represent the values in the elements of x. These strings are concise, single-line strings suitable for incorporation into multi-element output. If x is a cell, each element cell's contents are displayed, instead of each cell.

Unlike DISPSTR, DISPSTRS returns output describing each element of the input array individually.

This is used for constructing display output for functions like DISP. User-defined objects are expected to override DISPSTRS to produce suitable, readable output.

The output is human-consumable text. It does not have to be fully precise, and does not have to be parseable back to the original input. Full type information will not be inferrable from DISPSTRS output. The primary audience for DISPSTRS output is Octave programmers and advanced users.

The intention is for user-defined classes to override this method, providing customized display of their values.

The input x may be a value of any type. The main DISPSTRS implementation has support for Octave built-ins and common types. Other user-defined objects are displayed in a generic "m-by-n <class> array" format.

Returns a cellstr the same size as x.

Options: None are currently defined. This argument is reserved for future use.

Examples: dispstrs(magic(3))

See also: DISPSTR

6.2.3 errords

`errords (fmt, varargin)` [Function]
`errords (errorId, fmt, varargin)` [Function]

A variant of error() that supports dispstr functionality.

This is just like Octave's error(), except you can pass objects directly to `%s` conversion specifiers, and they will be automatically converted using dispstr.

6.2.4 fprintfds

`fprintfds (fmt, varargin)` [Function]
`fprintfds (fid, fmt, varargin)` [Function]

A variant of fprintf() that supports dispstr functionality.

This is just like Octave's fprintf(), except you can pass objects directly to `%s` conversion specifiers, and they will be automatically converted using dispstr.

See the documentation for SPRINTFDS for details on how it works.

Examples:

```
bday = Birthday(3, 14);
fprintfds('The value is: %s', bday)
```

See also: SPRINTFDS

6.2.5 logger.debug

`logger.debug (`*`msg`*`, `*`varargin`*`)` [Function]
`logger.debug (`*`exception`*`, `*`msg`*`, `*`varargin`*`)` [Function]

Log a DEBUG level message from caller, with printf style formatting.

This accepts a message with printf style formatting, using '%...' formatting controls as placeholders.

Examples:

```
logger.debug('Some message. value1=%s value2=%d', 'foo', 42);
```

6.2.6 logger.debugj

`logger.debugj (`*`msg`*`, `*`varargin`*`)` [Function]

Log a DEBUG level message from caller, with SLF4J style formatting.

This accepts a message with SLF4J style formatting, using '{}' as placeholders for values to be interpolated into the message.

Examples:

```
logger.debugj('Some message. value1={} value2={}', 'foo', 42);
```

6.2.7 logger.error

`logger.error (`*`msg`*`, `*`varargin`*`)` [Function]
`logger.error (`*`exception`*`, `*`msg`*`, `*`varargin`*`)` [Function]

Log an ERROR level message from caller, with printf style formatting.

This accepts a message with printf style formatting, using '%...' formatting controls as placeholders.

Examples:

```
logger.error('Some message. value1=%s value2=%d', 'foo', 42);
```

6.2.8 logger.errorj

`logger.errorj (`*`msg`*`, `*`varargin`*`)` [Function]

Log an ERROR level message from caller, with SLF4J style formatting.

This accepts a message with SLF4J style formatting, using '{}' as placeholders for values to be interpolated into the message.

Examples:

```
logger.errorj('Some message. value1={} value2={}', 'foo', 42);
```

6.2.9 logger.info

`logger.info (`*`msg`*`, `*`varargin`*`)` [Function]
`logger.info (`*`exception`*`, `*`msg`*`, `*`varargin`*`)` [Function]

Log an INFO level message from caller, with printf style formatting.

This accepts a message with printf style formatting, using '%...' formatting controls as placeholders.

Examples:

```
logger.info('Some message. value1=%s value2=%d', 'foo', 42);
```

6.2.10 logger.infoj

`logger.infoj (`*`msg`*`, `*`varargin`*`)` [Function]

Log an INFO level message from caller, with SLF4J style formatting.

This accepts a message with SLF4J style formatting, using '{}' as placeholders for values to be interpolated into the message.

Examples:

```
logger.infoj('Some message. value1={} value2={}', 'foo', 42);
```

6.2.11 logger.initSLF4M

`logger.initSLF4M ()` [Function]

SLF4M compatibility shim for initSLF4O

This is an alias for initSLF4O. Calling it just results in initSLF4O() being called.

This function is provided as a compatibility shim so that code which is expecting SLF4M will still work with SLF4O.

6.2.12 logger.initSLF4O

`logger.initSLF4O ()` [Function]

Initialzie SLF4O

This function must be called once before you use SLF4O.

6.2.13 logger.Log4jConfigurator

`logger.Log4jConfigurator` [Class]

A configurator tool for Log4j

This class configures the logging setup for Octave/SLF4O logging. It configures the log4j library that SLF4O logging sits on top of.

This class is provided as a convenience. You can also configure SLF4O logging by directly configuring log4j using its normal Java interface.

SLF4O does not automatically configure log4j. You must either call a configureXxx method on this class or configure log4j directly yourself to get logging to work. Otherwise, you may get warnings like this at the console:

log4j:WARN No appenders could be found for logger (unknown). log4j:WARN Please initialize the log4j system properly.

If that happens, it means you need to call logger.Log4jConfigurator.configureBasicConsoleLogging.

This also provides a log4j configuration GUI that you can launch with 'logger.Log4jConfigurator.showGui'.

Examples:

```
logger.Log4jConfigurator.configureBasicConsoleLogging

logger.Log4jConfigurator.setLevels({'root','DEBUG'});

logger.Log4jConfigurator.setLevels({
    'root'    'INFO'
    'net.apjanke.logger.swing'  'DEBUG'
    });

logger.Log4jConfigurator.prettyPrintLogConfiguration

% Display fully-qualified class/category names in the log output:
logger.Log4jConfigurator.setRootAppenderPattern(...
   ['%d{HH:mm:ss.SSS} %p %c - %m' sprintf('\n')]);

% Bring up the configuration GUI
logger.Log4jConfigurator.showGui
```

6.2.13.1 logger.Log4jConfigurator.configureBasicConsoleLogging

`logger.Log4jConfigurator.configureBasicConsoleLogging ()` [Static Method]

Configures log4j to do basic logging to the console

This sets up a basic log4j configuration, with log output going to the console, and the root logger set to the INFO level.

This method can safely be called multiple times. If there's already an appender on the root logger (indicating logging has already been configured), it silently does nothing.

6.2.13.2 logger.Log4jConfigurator.setRootAppenderPattern

`logger.Log4jConfigurator.setRootAppenderPattern (pattern)` [Static Method]

Sets the pattern on the root appender

This is just a convenience method. Assumes there is a single appender on the root logger.

6.2.13.3 logger.Log4jConfigurator.getLog4jLevel

`logger.Log4jConfigurator.getLog4jLevel (levelName)` [Static Method]

Gets the log4j Level Java enum value for a named level.

levelName is a charvec containing the name of the log level, such as `'INFO'` or `'DEBUG'`. It may also be one of the special names `'OFF'` or `'ALL'`.

Returns a Java org.apache.log4j.Level enum object.

6.2.13.4 logger.Log4jConfigurator.setLevels

`logger.Log4jConfigurator.setLevels (`*levels*`)` [Static Method]

Set the logging levels for multiple loggers

logger.Log4jConfigurator.setLevels(levels)

This is a convenience method for setting the logging levels for multiple loggers.

The levels input is an n-by-2 cellstr with logger names in column 1 and level names in column 2.

Examples:

```
logger.Log4jConfigurator.setLevels({'root','DEBUG'});

logger.Log4jConfigurator.setLevels({
    'root'    'INFO'
    'net.apjanke.logger.swing'  'DEBUG'
    });
```

6.2.13.5 logger.Log4jConfigurator.prettyPrintLogConfiguration

`logger.Log4jConfigurator.prettyPrintLogConfiguration ()` [Static Method]

`logger.Log4jConfigurator.prettyPrintLogConfiguration (`*verbose*`)` [Static Method]

Displays the current log configuration to the console.

verbose is a logical flag indicating whether verbose mode should be used. Defaults to false.

6.2.13.6 logger.Log4jConfigurator.showGui

`logger.Log4jConfigurator.showGui ()` [Static Method]

Display the Log4j configuration GUI provided by SLF4O.

BROKEN!!!

This tool is currently broken, and will probably crash Octave if you call it.

6.2.14 logger.Logger

`logger.Logger` [Class]

Main entry point through which logging happens

The Logger class provides method calls for performing logging, and the ability to look up loggers by name. This is the main entry point through which all SLF4O logging happens.

Usually you don't need to interact with this class directly, but can just call one of the error(), warn(), info(), debug(), or trace() functions in the logger namespace. Those will log messages using the calling class's name as the name of the logger. Also, don't call the constructor for this class. Use the static getLogger() method instead.

Use this class directly if you want to customize the names of the loggers to which logging is directed.

Each of the logging methods - error(), warn(), info(), debug(), and trace() - takes a sprintf()-style signature, with a format string as the first argument, and substitution values as the remaining arguments.

```
logger.info(format, varargin)
```

You can also insert an MException object at the beginning of the argument list to have its message and stack trace included in the log message.

```
logger.warn(exception, format, varargin)
```

See also: logger.error logger.warn logger.info logger.debug logger.trace

Examples:

```
log = logger.Logger.getLogger('foo.bar.FooBar');
log.info('Hello, world! Running on Octave %s', version);
try
    some_operation_that_could_go_wrong();
catch err
    log.warn(err, 'Caught exception during processing')
end
```

6.2.14.1 logger.Logger.getLogger

obj = logger.Logger.getLogger (*identifier*) [Static Method]

Gets the named Logger.

Returns a logger.Logger object.

6.2.14.2 logger.Logger.Logger

obj = logger.Logger (*jLogger*) [Constructor]

Build a new logger object around an SLF4J Logger object.

Generally, you shouldn't call this. Use logger.Logger.getLogger() instead.

6.2.14.3 logger.Logger.error

error (*obj*, *msg*, *varargin*) [Method]
error (*obj*, *exception*, *msg*, *varargin*) [Method]

Log a message at the ERROR level.

6.2.14.4 logger.Logger.warn

warn (*obj*, *msg*, *varargin*) [Method]
warn (*obj*, *exception*, *msg*, *varargin*) [Method]

Log a message at the WARN level.

6.2.14.5 logger.Logger.info

info (*obj*, *msg*, *varargin*) [Method]
info (*obj*, *exception*, *msg*, *varargin*) [Method]

Log a message at the INFO level.

6.2.14.6 logger.Logger.debug

`debug (obj, msg, varargin)` [Method]
`debug (obj, exception, msg, varargin)` [Method]
Log a message at the DEBUG level.

6.2.14.7 logger.Logger.trace

`trace (obj, msg, varargin)` [Method]
`trace (obj, exception, msg, varargin)` [Method]
Log a message at the TRACE level.

6.2.14.8 logger.Logger.errorj

`errorj (obj, msg, varargin)` [Method]
Log a message at the ERROR level, using SLF4J formatting.

6.2.14.9 logger.Logger.warnj

`warnj (obj, msg, varargin)` [Method]
Log a message at the WARN level, using SLF4J formatting.

6.2.14.10 logger.Logger.infoj

`infoj (obj, msg, varargin)` [Method]
Log a message at the INFO level, using SLF4J formatting.

6.2.14.11 logger.Logger.debugj

`debugj (obj, msg, varargin)` [Method]
Log a message at the DEBUG level, using SLF4J formatting.

6.2.14.12 logger.Logger.tracej

`tracej (obj, msg, varargin)` [Method]
Log a message at the TRACE level, using SLF4J formatting.

6.2.14.13 logger.Logger.isErrorEnabled

`out = isErrorEnabled (obj)` [Method]
True if ERROR level logging is enabled for this logger.

6.2.14.14 logger.Logger.isWarnEnabled

`out = isWarnEnabled (obj)` [Method]
True if WARN level logging is enabled for this logger.

6.2.14.15 logger.Logger.isInfoEnabled

`out = isInfoEnabled (obj)` [Method]
True if INFO level logging is enabled for this logger.

6.2.14.16 logger.Logger.isDebugEnabled

`out = isDebugEnabled (obj)` [Method]

True if DEBUG level logging is enabled for this logger.

6.2.14.17 logger.Logger.isTraceEnabled

`out = isTraceEnabled (obj)` [Method]

True if TRACE level logging is enabled for this logger.

6.2.14.18 logger.Logger.listEnabledLevels

`out = listEnabledLevels (obj)` [Method]

List the levels that are enabled for this logger.

The enabled levels are listed by name.

Returns a cellstr vector or empty.

6.2.15 logger.trace

`logger.trace (msg, varargin)` [Function]
`logger.trace (exception, msg, varargin)` [Function]

Log a TRACE level message from caller, with printf style formatting.

This accepts a message with printf style formatting, using '%...' formatting controls as placeholders.

Examples:

```
logger.trace('Some message. value1=%s value2=%d', 'foo', 42);
```

6.2.16 logger.tracej

`logger.tracej (msg, varargin)` [Function]

Log a TRACE level message from caller, with SLF4J style formatting.

This accepts a message with SLF4J style formatting, using '{}' as placeholders for values to be interpolated into the message.

Examples:

```
logger.tracej('Some message. value1={} value2={}', 'foo', 42);
```

6.2.17 logger.version

`logger.version ()` [Function]
`out = logger.version ()` [Function]

Gets version info for the SLF4O library.

If return value is not captured, displays version info for SLF4O and related libraries to the console.

If return value is captured, returns the version of the SLF4O library as a char vector.

6.2.18 logger.warn

`logger.warn (`*`msg`*`, `*`varargin`*`)` [Function]
`logger.warn (`*`exception`*`, `*`msg`*`, `*`varargin`*`)` [Function]

Log a WARN level message from caller, with printf style formatting.

This accepts a message with printf style formatting, using '%...' formatting controls as placeholders.

Examples:

```
logger.warn('Some message. value1=%s value2=%d', 'foo', 42);
```

6.2.19 logger.warnj

`logger.warnj (`*`msg`*`, `*`varargin`*`)` [Function]

Log a WARN level message from caller, with SLF4J style formatting.

This accepts a message with SLF4J style formatting, using '{}' as placeholders for values to be interpolated into the message.

Examples:

```
logger.warnj('Some message. value1={} value2={}', 'foo', 42);
```

6.2.20 mustBeA

`mustBeA (`*`value`*`, `*`type`*`)` [Function]

Validate that an input is of a particular data type.

Validates that the input Value is of the specified Type or a subtype. If Value is not of Type, an error is raised. If Value is of Type, does nothing and returns.

Value is the value to validates the type of. It may be anything. If you call it using a variable (as opposed to a longer expression), the variable name is included in any error messages.

Type (char) is the name of the type that Value must be. A type name may be one of: * A class, such as 'double', 'cell', or 'containers.Map' * One of the special SLF4O pseudotypes: cellstr numeric object any

Note: The cellstr pseudotype is nontrivial to check for, as it must call iscellstr() and check all cell contents.

Examples:

```
function foo(x, someStrings)
  mustBeA(x, 'double');
  mustBeA(someStrings, 'cellstr');
endfunction
```

6.2.21 pp

`pp (`*`varargin`*`)` [Function]

Command wrapper for prettyprint().

PP is a command-oriented wrapper for prettyprint, intended for interactive use. Code should not call this.

Right now, it just calls PRETTYPRINT on its input, but the intention is to extend it to take variable names as chars in addition to the normal prettyprint input, so you can say 'pp foo' instead of 'pp(foo)'. This is purely a convenience to save users from typing in parentheses.

6.2.22 prettyprint

out = `prettyprint` (*x*) [Function]

Formatted output of array contents.

Displays a formatted, human-readable representation of the contents of a value. This is a detailed, multi-line output that typically displays all the individual values in an array, or drills down one or more levels into complex objects. In many cases, this is just like doing a DISP, but it respects the DISPSTR and DISPSTRS methods defined on user-defined objects inside complex types, where DISP does not.

This output is for human consumption and its format may change over time. The format may also be dependent on settings in the Octave session, such as 'format' and the user's locale.

The default PRETTYPRINT implementation has support for Octave built-in types, structs, cells, and tables, and, unlike the default disp() behavior for them, respects DISPSTRS defined for values inside structs, cells, and tables.

The input x may be a value of any type.

If the output is not captured, displays its results to the console. If the output is captured, returns its results as char.

The intention is for user-defined classes to override this method, providing customized display of their values.

6.2.23 size2str

out = `size2str` (*sz*) [Function]

Format a matrix size for display.

Sz is an array of dimension sizes, in the format returned by SIZE.

Returns a charvec.

6.2.24 sprintfds

`sprintfds` (*fmt*, *varargin*) [Function]

A variant of sprintf() that supports dispstr functionality.

This is just like Octave's sprintf(), except you can pass objects directly to `%s` conversion specifiers, and they will be automatically converted using dispstr.

For inputs that are objects, dispstr() is implicitly called on them, so you can pass them directly to '%s' conversion specifiers in your format string.

Examples:

```
bday = Birthday(3, 14);
str = sprintfds('The value is: %s', bday)
```

See also: FPRINTFDS

6.2.25 warningds

`warningds (`*`fmt`*`, `*`varargin`*`)` [Function]
`warningds (`*`warningId`*`, `*`fmt`*`, `*`varargin`*`)` [Function]

A variant of warning() that supports dispstr functionality.

This is just like Octave's warning(), except you can pass objects directly to `%s` conversion specifiers, and they will be automatically converted using dispstr.

7 Copying

7.1 Package Copyright

SLF4O for Octave is covered by the GNU GPLv3.

All the code in the package is GNU GPLv3.

7.2 Manual Copyright

This manual is for SLF4O, version 0.1.0-SNAPSHOT.