Skookum JS Documentation

Release 0.0.1

Saúl Ibarra Corretgé

Jun 01, 2018

Contents

Skookum JS, or *sjs* for short, is a JavaScript runtime built on top of the Duktape engine. It provides a simple way to write applications using JavaScript with a traditional synchronous I/O model. It tries to minimize abstractions and give the developer as much control as possible over the platform and low level APIs.

CHAPTER 1

Quickstart

```
git clone https://github.com/saghul/sjs
cd sjs
make
./build/sjs
```

CHAPTER 2

Documentation

2.1 Features

- Small footprint
- Ecmascript ES5/ES5.1 compliant
- Some post-ES5 features
- TypedArray and Buffer support
- Built-in regular expression engine
- Built-in Unicode support
- Tail call support
- Combined reference counting and mark-and-sweep garbage collection with finalization
- CommonJS-based module loading system
- Support for native modules written in C
- Rich standard library
- Binary name 25% shorter than Node

See also:

Skookum JS gets most of its features through its engine: Duktape.

2.1.1 Post-ES5 features

Duktape (the JavaScript engine used by *sjs*) currently implements some post-ES5 features.

sjs expands on this by having babel-polyfill builtin.

Note: Some of the polyfills depend on core dunctionality not currently present, such as `setTimeout`.

2.2 Design

Skookum JS is build by abstracting the undelying platform as little as possible and giving the developer full control of low level POSIX APIs.

Note: In release `18.6.0` SkookumJS went through a major internal redesign. The old design is still documented in this page, for posterity.

The SkookumJS VM is inspired by Node.js in the following ways:

- A single binary is provided, which contains all modules already builtin.
- Binary addons can be created, which can be linked with the `sjs` binary.

In addition, a `libsjs` library target is available for other applications to link with / embed.

2.2.1 Implementation details

- The Duktape engine provides JavaScript capabilities.
- The REPL (interactive mode) uses linenoise.
- Modules can be either written in pure JavaScript or in C. See the *Modules* section.

2.2.2 Retrospective

After first implementing the old design, I came to realization that it's more troublesome than the current one. Specifically, these are issues that promted the redesign:

- Cumbersome fiddling with `SJS_PATH` to make sure the built binary and tests had access to modules.
- Complex build system with many shared libraries.
- Hard to have multiple versions side by side.
- Potential standard library / application mismatch.

In retrospect this should have been the design from the get go. It still offers the same amount of flexibility (binary addons, *libsjs*) while making it easier to use and build.

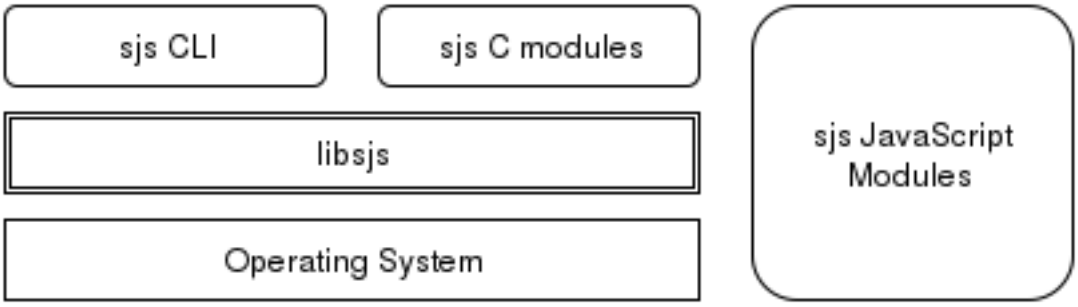
2.2.3 Design (old)

The VM is modeled after other runtimes such as Ruby's or Python's.

The runtime can be logically divided into 3 parts:

- The interpreter CLI: it's a relatively simple application which creates a *sjs* VM and evaluates code using it. For the REPL (interactive mode) it uses linenoise.

- The VM library: *sjs* (the CLI) links with *libsjs* (the VM), which encapsulates all functionality and embed the Duktape engine. This makes other applications capable of running JavaScript code by linking with *libsjs* and creating a VM.
- The modules: in *sjs* modules can be either written in pure JavaScript or in C. See the *Modules* section.



The VM

The *sjs* VM is currently single threaded, bu thread aware. That is: users can create as many VMs as they wish, as long as there is a single VM running for a given thread. There is no global state. There is no builtin mechanism for inter-VM communication.

See the *Embedding sjs in your application* section for details on the C API.

Inspiration

Skookum JS was not inspired by browser JavaScript runtimes but by more "traditional" runtimes such as Ruby's MRI or Python's CPython.

This means that the model is not inherently asynchronous and event-driven, as in a browser or Node JS. *sjs* does provide the necessary tools to build asynchronous / event-driven frameworks on top of the provided modules.

2.3 Usage

Following its design, *sjs* can be used in 2 ways: the CLI and the C API.

2.3.1 CLI

This is the typical way to run JavaScript applications with *sjs*, by using the `sjs` binary.

```
sjs -h
Usage: sjs [options] [ <code> | <file> | - ]

-h          show help text
-i          enter interactive mode after executing argument file(s) / eval code
-e CODE     evaluate code

If <file> is omitted, interactive mode is started automatically.
```

With the *sjs* CLI you can execute a file, eval some code directly from the command line, or enter the interactive mode.

Note: When using the CLI *sjs* will try to pretify the output of every command by serializing it using a JSON variant called JX which Duktape includes.

2.3.2 Embedding sjs in your application

The *sjs* VM API allows applications to execute JavaScript code using the *sjs* engine. The prime example is the *sjs* CLI, see `src/cli/main.c`.

With this C API applications willing to run JavaScript code can embed the *sjs* engine or link with *libsjs*.

Data types

`sjs_vm_t`

Opaque type encapsulating the *sjs* VM.

`duk_context`

Opaque type encapsulating the Duktape engine.

API

Main VM API

*sjs_vm_t** **`sjs_vm_create`**(void)
Create a *sjs* VM.

Returns The initialized VM.

void **`sjs_vm_destroy`**(*sjs_vm_t* *vm*)
Destroy the given VM. It can no longer be used after calling this function.

Parameters

- **`vm`** – The VM which will be destroyed.

void **`sjs_vm_setup_args`**(*sjs_vm_t* *vm*, int *argc*, char* *argv[]*)
Setup the VM's arguments. This is required for initializing some internals in the *system* module.

Parameters

- **`vm`** – The VM reference.
- **`argc`** – Number of arguments in the array.
- **`argv`** – Array with arguments, in comman-line form.

*duk_context** **`sjs_vm_get_duk_ctx`**(*sjs_vm_t* *vm*)
Get the reference to the Duktape engine associated with the VM.

Parameters

- **`vm`** – The VM reference.

Returns The Duktape context.

*sjs_vm_t** **`sjs_vm_get_vm`**(*duk_context* *ctx*)
Get the reference to the *sjs* VM associated with the given Duktape context.

Parameters

- **ctx** – The Duktape context.

Returns The *sjs* VM instance.

int **sjs_vm_eval_code**(const *sjs_vm_t** *vm*, const char* *filename*, const char* *code*, size_t *len*, FILE* *foutput*, FILE* *ferror*)
Evaluate the given JavaScript *code*. The code is wrapped in a CommonJS module function and executed.

Parameters

- **vm** – The VM reference.
- **filename** – Indicates the filename that is being executed. It will be printed in tracebacks and such.
- **code** – What is going to be executed.
- **len** – Length of the code.
- **foutput** – Stream where to print the result of the evaulated code (can be NULL).
- **ferror** – Stream where to print errors, if any (can be NULL).

Returns 0 if the code was evaluated without errors, != 0 otherwise.

int **sjs_vm_eval_code_global**(const *sjs_vm_t** *vm*, const char* *filename*, const char* *code*, size_t *len*, FILE* *foutput*, FILE* *ferror*)
Similar to *sjs_vm_eval_code()* but it evaluates the code in the global scope instead of creating a new CommonJS style context.

int **sjs_vm_eval_file**(const *sjs_vm_t** *vm*, const char* *filename*, FILE* *foutput*, FILE* *ferror*)
Evaluate the given file as JavaScript code. The code is wrapped in a CommonJS module function and executed.

Parameters

- **vm** – The VM reference.
- **filename** – The file to be evaluated.
- **foutput** – Stream where to print the result of the evaulated code (can be NULL).
- **ferror** – Stream where to print errors, if any (can be NULL).

Returns 0 if the code was evaluated without errors, != 0 otherwise.

Utility functions

int **sjs_path_normalize**(const char* *path*, char* *normalized_path*, size_t *normalized_path_len*)
Normalize the given *path* into the given buffer. Mormalizing a path includes tilde expansions and realpath(3).

Parameters

- **path** – The path which needs to be normalized.
- **normalized_path** – Buffer to store the normalized path.
- **normalized_path_len** – Size of *normalized_path*.

Returns 0 on success, or < 0 on failure. The returned code is the negated *errno*.

int **sjs_path_expanduser**(const char* *path*, char* *normalized_path*, size_t *normalized_path_len*)
Similar to *sjs_path_normalize()* but in only performs tilde expansion.

2.4 The module system

sjs uses a CommonJS compliant module system, also inspired by the Node module system.

2.4.1 Loading modules

Modules are loaded using the *require()* function. Given a *module id*, it returns the module exported data.

The module id can be a relative path or a non-relative path, in which case the module is loaded from the system search directories. See *Module search paths*.

2.4.2 Module context

When a module is being loaded the following *module globals* are available:

__filename
Path to the file being executed. In the global scope it constains the filename which is currently being executed, <repl> if running in the REPL, <stdin> if code is being read from *stdin*, or <eval> if evaluating code straight from the CLI. Inside a module, it contains the absolute path to the module file.

__dirname
Directory where the file being evaluated is, obtained by applying dirname(3) over __filename.

module
The current *Module()* object instance.

exports
The current module exports. It's a reference to module.exports.

require(*id*)
Loads the requested module and returns the module's exports.

Example, assuming some foo.js file with the following content:

```
function foo() {
    return 42;
}

exports.foo = foo;
```

It can be loaded and used as follows:

```
const mod = require('./foo');

print(mod.foo());
// prints 42
```

The Module object

class Module()
Object representing a JavaScript module (for the lack of a better term).

Module.Module.**filename**

Fully resolved filename of the module.

Module.Module.**id**

Same as filename.

Module.Module.**loaded**

Boolean attribute indicated if the module was loaded or if it's in the process of being loaded.

Module.Module.**exports**

Object containing the functions and attributes to be exported.

The "main" module

The *require()* function has a `main` attribute, referencing the current module only for the "main" module. Otherwise it's `undefined`.

The following construct can be used in order to differentiate if a module was require()-d or directly run:

```
if (require.main === module) {
    // module was directly run
}
```

2.4.3 Module search paths

Modules are located by their *module id*. This module id can be one of:

- a relative path: ex. `./foo` or `../foo`
- a regular module id: ex. `system`

Absolute paths are not supported.

Relative paths are resolved relative to the *calling* module, or the module which contains the call to *require()*.

Regular module ids are resolved by looking into the *system* paths in `system.path`. The list of paths to search for modules is dynamic and can be modified at runtime. The following are the builtin system paths:

- `/usr/lib/sjs/modules`
- `/usr/local/lib/sjs/modules`
- `~/.local/sjs/modules`

2.5 Modules

sjs includes a builtin Common JS module loading system. See *The module system*. There is also a *Standard library* whith many modules. The goal is to provide a *kitchen-sink* collection of modules.

Starting with version 0.4.0, *sjs* includes a module system more similar to Node.

2.5.1 Standard library

assert

This module is primarily used to write tests. It originates from this commonjs-assert module.

Functions

assert.**fail**(*value*, *expected*, *message*, *operator*)

Throws an exception that displays the values for *value* and *expected* separated by the provided *operator*. If *message* is undefined, "value operator expected" is used.

`assert.ok` (*value*[, *message*])
Thests if *value* is truthy.

`assert.equal` (*value*, *expected*[, *message*])
Tests shallow, coercive equality with the equal comparison operator (==).

`assert.notEqual` (*value*, *expected*[, *message*])
Tests shallow, coercive non-equality with the not equal comparison operator (!=).

`assert.deepEqual` (*value*, *expected*[, *message*])
Tests for deep equality.

`assert.notDeepEqual` (*value*, *expected*[, *message*])
Tests for any deep inequality.

`assert.strictEqual` (*value*, *expected*[, *message*])
Tests strict equality, as determined by the strict equality operator (===).

`assert.notStrictEqual` (*value*, *expected*[, *message*])
Tests strict non-equality, as determined by the strict not equal operator (!==).

`assert.throws` (*block*[, *error*][, *message*])
Expects *block* to throw an error. *error* can be constructor, RegExp or validation function.

Validate instanceof using constructor:

```
assert.throws(function() { throw new Error("Wrong value"); }, Error);
```

Validate error message using RegExp:

```
assert.throws(function() { throw new Error("Wrong value"); }, /value/);
```

Custom error validation:

```
assert.throws(function() {
    throw new Error("Wrong value");
}, function(err) {
    if ( (err instanceof Error) && /value/.test(err) ) {
        return true;
    }
}, "unexpected error");
```

`assert.doesNotThrow` (*block*[, *message*])
Expects *block* not to throw an error, see *assert.throws()* for details.

codecs

The *codecs* module provides a consistent API for encoding strings (or objects) into different representations. The following encodeings are supported:

- `base64`
- `hex`
- `json`
- `punycode`
- `utf8`

encode (*encoding*, *data*)
Encodes the given *data* using the desired *encoding*.

Arguments

- **encoding** – The desired encoding. (Must be one of the supported encodings)
- **data** – Value which will be encoded.

Returns A string with the encoded data.

decode(*encoding*, *data*)
Decodes the given *data* using the desired *encoding*.

Arguments

- **encoding** – The desired encoding. (Must be one of the supported encodings)
- **data** – Value which will be decoded.

Returns The decoded object.

console

This module provides well known logging facilities for JavaScript programs. This module (partially) implements the CommonJS Console specification.

Console object

The *Console* object encapsulates all functionality in the *console* module in the form of a class. The module exports a default instance of this class though more can be created (which is rarely needed).

class console.**Console**(*stdout*[, *stderr*])
Create an instance of a console object.

Arguments

- **stdout** – Object representing standard out. It must implement the same API as *io.File()*.
- **stderr** – Analogous to *stdout*, but representing the standard error. If omitted *stdout* is used.

console.Console.console.Console.prototype.**assert**(*expression*[, *message*][, ...])
Simple assertion test using the *assert* module. The given *expression* is evaluated and if it's falsey the given *message* is given with an `AssertionError`.

```
console.assert(false, 'Whoops %s', 'Houston, we\'ve got a problem!');
// AssertionError: Whoops Houston, we've got a problem!
```

console.Console.console.Console.prototype.**log**()
Outpits the given data to `stdout` with a newline. Data is formatted using *utils.object.format()*.

```
console.log('Hello %s! Here is a number: %d and an object: %j', 'sjs', 42,
↪{foo: 'foo', bar: 'bar'})
// Hello sjs! Here is a number: 42 and an object: {"foo":"foo","bar":"bar"}
```

console.Console.console.Console.prototype.**info**()
Alias for `console.Console.prototype.log()`.

console.Console.console.Console.prototype.**error**()
Similar to `console.Console.prototype.log()` but it outputs to `stderr`.

console.Console.console.Console.prototype.**warn**()
Alias for console.Console.prototype.error().

console.Console.console.Console.prototype.**dir**(*obj*[, *options*])
Inspect the given object using :js:func:‘‘ and print output to stdout.

```
console.dir(console);
// { log: [Function],
//   info: [Function],
//   warn: [Function],
//   error: [Function],
//   dir: [Function],
//   time: [Function],
//   timeEnd: [Function],
//   trace: [Function],
//   assert: [Function],
//   Console: [Function: Console] }
```

Note: The *customInspect* option is ignored if given.

console.Console.console.Console.prototype.**time**(*label*)

console.Console.console.Console.prototype.**timeEnd**(*label*)
Create a labeled timestamp. When timeEnd is called the time difference is computed and printed to staout.

```
console.time('foo');
// (wait some time...)
console.timeEnd('foo');
// foo: 6535.996ms
```

console.Console.console.Console.prototype.**trace**([*message*][, ...])
Prints a stack trace at the current position to stderr. If an optional message and formatting options are given the message is formatted using *utils.object.format()*.

```
console.trace();
// Trace
//     at global (<repl>:1) preventsyield
```

Functions

console.**assert**()

console.**log**()

console.**info**()

console.**warn**()

console.**error**()

console.**dir**()

console.**time**()

console.**timeEnd**()

console.**trace**()

Functions bound to the default *console.Console()* instance.

errno

The *errno* module exposes all system error codes matching reason strings.

Note: The exposed error codes may vary depending on the platform.

errno.**map**
A Map mapping error codes to their string versions. Example:

```
sjs> errno.map.get(1)
= EPERM
```

errno.**E***
All errno constants available in the system as exposed as module level constants. Example:

```
sjs> errno.EPERM
= 1
```

errno.**strerror**(*code*)
Get the string that describes the given error *code*.

hash

This module provides facilities for creating varios hashes over data.

Hash objects

All objects defined in this module inherit from a HashBase base class and thus have the same API.

class hash.**HashBase**()
Base class for all hash types. Currently the following classes are implemented:

- MD5
- SHA1
- SHA256
- SHA512

hash.HashBase.hash.HashBase.prototype.**update**(*data*)
Update the hash object with the given data. This function can be called multiple times in order to incrementally update the data being hashed.

Functions

hash.**createHash**(*type*)
Helper function to create hash objects. type must be (a string) one of:

- md5
- sha1

- sha256
- sha512

Examples

Using the helper function and chaining methods:

```
hash.createHash('sha1').update('abc').digest('hex');
// "a9993e364706816aba3e25717850c26c9cd0d89d"
```

Creating objects and calling methods individually:

```
var h1 = new hash.SHA1();
h1.update('abc');
h1.digest('hex');
// "a9993e364706816aba3e25717850c26c9cd0d89d"
```

io

This module provies access to high i/o primitives and streams.

File object

class io.**File**()
Class representing a stdio stream.

This class created through *io.open()* or *io.fdopen()*, never directly.

io.File.**path**
Returns the opened file's path.

io.File.**fd**
Returns the file descriptor associated with the file.

io.File.**mode**
Returns the mode in which the file was opened.

io.File.**closed**
Boolean flag indicating if the file was closed.

io.FIle.prototype.**read**(*nread*)
Read data from the file.

Arguments

- **nread** – Amount of data to receive. If not specified it defaults to 4096. Alternatively, a *Buffer* can be passed, and data will be read into it.

Returns The data that was read as a *Uint8Array* or the amount of data read as a number, if a *Buffer* was passed.

See also:

fread(3)

`io.FIle.prototype.`**`readLine`**`(`*nread*`)`
Similar to *io.FIle.prototype.read()*, but stops at the newline (`\n`) character. This is the recommended function to read from *stdin*, but not from binary files.

See also:

fgets(3)

`io.FIle.prototype.`**`write`**`(`*data*`)`
Write data on the file.

Arguments

- **`data`** – The data that will be written (can be a string or a *Buffer*).

Returns The number of bytes from *data* which were actually written.

See also:

fwrite(3)

`io.FIle.prototype.`**`writeLine`**`(`*data*`)`
Same as *io.FIle.prototype.write()*, but add a newline (`\n`) at the end.

`io.FIle.prototype.`**`flush`**`()`
Flush the buffered write data to the file.

See also:

fflush(3)

`io.FIle.prototype.`**`close`**`()`
Close the file.

Functions

`io.`**`open`**`(`*path*, *mode*[, *buffering*]`)`
Opens the file at the given *path* in the given mode. Check fopen(3) for the *mode* details. It returns a *io.File()* object.

If *buffering* is specified, it must be `-1` (for default buffering), `0` (for unbuffeered) or `1` for line buffering). See setvbuf(3).

`io.`**`fdopen`**`(`*fd*, *mode*[, *path*][, *buffering*]`)`
Opens the fiven file descriptor in *fd* as a *io.File()* object. The given *mode* must be compatible with how the file descriptor was opened. *path* is purely informational.

If *buffering* is specified, it must be `-1` (for default buffering), `0` (for unbuffeered) or `1` for line buffering). See setvbuf(3).

See also:

fdopen(3)

`io.`**`readFile`**`(`*path*`)`
Returns the contents of the file at the given *path* as a *Uint8Array*.

io.select

This object provides access to select(2).

`select.`**`select`**(*rfds*, *wfds*, *xfds*, *timeout*)
Wait until any of the given file descriptors are ready for reading, writing or have a pending exceptional condition.

Arguments

- **`rfds`** – Array of file descriptors to monitor for reading.
- **`wfds`** – Array of file descriptors to monitor for writing.
- **`xfds`** – Array of file descriptors to monitor for pending exceptional conditions.
- **`timeout`** – Amount of time to wait. `null` means unlimited. This function might return early if interrupted by a signal.

Returns An object containing 3 properties: *rfds*, *wfds* and *xfds*, containing the file descriptors which are ready for each condition respectively.

For more information see select(2).

io.poll

This object provides access to poll(2).

`poll.`**`POLLIN`**

`poll.`**`POLLOUT`**

`poll.`**`POLLPRI`**

`poll.`**`POLLRDHUP`**

`poll.`**`POLLERR`**

`poll.`**`POLLHUP`**

`poll.`**`POLLINVAL`**
Constants to be used in the *events* or *revents* fields of a `pollfd` object. Check poll(2) for more information. Note that not all these constants might be available on your platform.

`poll.`**`poll`**(*pfds*, *timeout*)
Examines the given file descriptors to see if some of them are ready for i/o or if certain events have occurred on them.

Arguments

- **`pfds`** – An array of `pollfd` objects to be examined. A `pollfd` object is any object which has a *fd* and a *events* properties. The *events* property must contain the or-ed events that the user is interested in examining.
- **`timeout`** – Amount of time to wait. `null` means unlimited. This function might return early if interrupted by a signal.

Returns An array of `pollfd` objects, containing *fd*, *events* and *revents* properties. *fd* and *events* match the given ones, and *revents* indicates the received events.

For more information see poll(2).

net

This module provides access to networking utilities.

Socket object

The *Socket* object is a thin object oriented wrapper around socket(2) and its related functionality.

class net.**Socket**(*domain*, *type*[, *options*])
Creates a socket object.

Arguments

- **domain** – Socket domain (one of *net.AF_INET*, *net.AF_INET6* or *net.AF_UNIX*).
- **type** – Socket type (one of *net.SOCK_STREAM* or *net.SOCK_DGRAM*).
- **options** – Optional object which may contain the following properties:
  - *fd*: if specified, the socket is built with the given fd, an a new one is not created.
  - *nonBlocking*: if specified and `true`, the socket will be breated in non-blocking mode.

See also:

socket(2)

Note: Since version 0.3.0 the fds are created with `O_CLOEXEC` set. You can undo this using *os.cloexec()*.

---

net.Socket.**fd**
Returns the file descriptor representing the socket.

net.Socket.prototype.**accept**()
Waits for an incoming connection and accepts it. Only works on listening sockets.

Returns The accepted *net.Socket()* object.

See also:

accept(2)

net.Socket.prototype.**bind**(*address*)
Bind the socket to the given *address*. See *Socket addresses* for the format.

See also:

bind(2)

net.Socket.prototype.**close**()
Closes the socket.

See also:

close(2)

net.Socket.prototype.**connect**(*address*)
Starts a connection towards the given *address*. See *Socket addresses* for the format.

See also:

connect(2)

net.Socket.prototype.**getsockname**()
Returns the socket's local address. See *Socket addresses* for the format.

See also:

getsockname(2)

net.Socket.prototype.**getpeername**()
Returns the socket's peer address. See *Socket addresses* for the format.

See also:

getpeername(2)

net.Socket.prototype.**listen**([*backlog*])
Set the socket in listening mode, ready for accepting connections.

Arguments

- **backlog** – The maximum length for the queue of pending connections. If not set it defaults to 128.

See also:

listen(2)

net.Socket.prototype.**recv**([*nrecv*])
Receive data from a socket. It can only be used in a connected socket.

Arguments

- **nrecv** – Maximum amount of data to receive. If not specified it defaults to 4096. Alternatively, a *Buffer* can be passed, and data will be read into it.

Returns The data that was read as a *Uint8Array* or the amount of data read as a number, if a *Buffer* was passed.

See also:

recv(2)

net.Socket.prototype.**send**(*data*)
Transmit a message to the other socket. It can only be used with connected sockets.

Arguments

- **data** – The message that will be transmitted (can be a string or a *Buffer*).

Returns The number of bytes from *data* which were actually sent.

See also:

send(2)

net.Socket.prototype.**recvfrom**([*nrecv*])
Similar to *net.Socket.prototype.recv()* but it can also be used in non-connected sockets.

Returns An object with 2 properties: *address*, which contains the address of the sender and *nread* if a *Buffer* was used, or *data*, with the data as a *Uint8Array*.

See also:

recvfrom(2)

net.Socket.prototype.**sendto**(*data*, *address*)
Similar to *net.Socket.prototype.recv()* but it can also be used in non-connected sockets and the destination address has to be specified.

See also:

sendto(2)

net.Socket.prototype.**shutdown**(*how*)
Causes all or part of a full-duplex connection to be shut down. *how* must be one of *net.SHUT_RD*, *net.SHUT_WR* or *net.SHUT_RDWR*.

See also:

shutdown(2)

net.Socket.prototype.**setsockopt**(*level*, *option*, *value*)
Set a socket option on the given *level*. *value* may contain either a number (for numeric or boolean options) or a string containing the binary representation of the value. Use of *Buffer* objects is recommended to build the binary value.

Example, setting a numeric or boolean option:

```
var sock = new net.Socket(net.AF_INET, net.SOCK_STREAM);
sock.setsockopt(net.SOL_SOCKET, net.SO_REUSEADDR, true);
```

Example, setting a binary option:

```
var sock = new net.Socket(net.AF_INET, net.SOCK_STREAM);
var lingerOpts = new Buffer(8);
lingerOpts.writeInt32LE(1, 0);    // enable lingering
lingerOpts.writeInt32LE(100, 4);    // linger for 100 seconds
sock.setsockopt(net.SOL_SOCKET, net.SO_LINGER, lingerOpts.toString());
```

See also:

setsockopt(2)

net.Socket.prototype.**getsockopt**(*level*, *option*[, *size*])
Get the value for a socket *option* on the given *level*. When *size* is omitted, the value is assumed to be an integer, but when a number is given a buffer of *size* size is used. A *Buffer* object can be used to parse the result.

Example, getting a numeric or boolean option:

```
var sock = new net.Socket(net.AF_INET, net.SOCK_STREAM);
sock.setsockopt(net.SOL_SOCKET, net.SO_REUSEADDR, true);
var r = sock.getsockopt(net.SOL_SOCKET, net.SO_REUSEADDR);
```

Example, getting a binary option:

```
var sock = new net.Socket(net.AF_INET, net.SOCK_STREAM);
var lingerOpts = new Buffer(8);
lingerOpts.writeInt32LE(1, 0);    // enable lingering
lingerOpts.writeInt32LE(100, 4);    // linger for 100 seconds
sock.setsockopt(net.SOL_SOCKET, net.SO_LINGER, lingerOpts.toString());
var r = sock.sgetsockopt(net.SOL_SOCKET, net.SO_LINGER, 8);
var resBuf = new Buffer(r);
assert.equal(resBuf.readInt32LE(0), 1)
assert.equal(resBuf.readInt32LE(4), 100)
```

See also:

getsockopt(2)

net.Socket.prototype.**setNonBlocking**(*set*)
Sets the socket in non-blocking mode if `true`, or blocking mode if `false`.

Constants

net.**AF_INET**
IPv4 socket domain.

net.**AF_INET6**
IPv6 socket domain.

net.**AF_UNIX**
Unix socket domain.

net.**SOCK_STREAM**
Stream socket type.

net.**SOCK_DGRAM**
Datagram socket type.

net.**SHUT_RD**

net.**SHUT_WR**

net.**SHUT_RDWR**
Shutdown modes for *net.Socket.prototype.shutdown()*.

Functions

Socket addresses

Throughout this module, when an address is taken as a parameter or returned from a function, it's expressed as an object with different properties, depending on the address family:

- IPv4 sockets (AF_INET family): object containing `host` and `port` properties.
- IPv6 sockets (AF_INET6 family): object containing `host`, `port`, `flowinfo` and `scopeid` properties. The last two can be omitted and will be assumed to be 0.
- Unix sockets (AF_UNIX family): string containing the path.

getaddrinfo

net.**getaddrinfo**(*hostname*, *servname*[, *hints*])
Get a list of IP addresses and port numbers for host *hostname* and service *servname*. See getaddrinfo(3) for details.

Example:

```
sjs> var r = net.getaddrinfo('google.com', 'http');
sjs> outil.inspect(r);
= [ { family: 2,
    type: 2,
    protocol: 17,
    canonname: '',
    address: { host: '216.58.198.174', port: 80 } },
{ family: 2,
    type: 1,
    protocol: 6,
    canonname: '',
```

(continues on next page)

(continued from previous page)

```
    address: { host: '216.58.198.174', port: 80 } },
{ family: 30,
    type: 2,
    protocol: 17,
    canonname: '',
    address:
    { host: '2a00:1450:4009:809::200e',
    port: 80,
    flowinfo: 0,
    scopeid: 0 } },
{ family: 30,
    type: 1,
    protocol: 6,
    canonname: '',
    address:
    { host: '2a00:1450:4009:809::200e',
    port: 80,
    flowinfo: 0,
    scopeid: 0 } } ]
```

The *hints* optional argument may contain an object with the following properties: `family`, `type`, `protocol` and `flags`.

Example:

```
sjs> var r = net.getaddrinfo('google.com', 'http', {family: net.AF_INET});
sjs> outil.inspect(r);
= [ { family: 2,
    type: 2,
    protocol: 17,
    canonname: '',
    address: { host: '216.58.198.174', port: 80 } },
{ family: 2,
    type: 1,
    protocol: 6,
    canonname: '',
    address: { host: '216.58.198.174', port: 80 } } ]
```

The returned result is a list of objects containing the following properties: `family`, `type`, `protocol`, `canonname` and `address`.

`net.`**`gai_strerror`**(*code*)
Get the string that describes the given error *code*.

`net.`**`gai_error_map`**
A `Map` mapping *getaddrinfo* error codes to their string versions.

`net.`**`AI_*`**
All *addrinfo* constants to be used as hints in *`net.getaddrinfo()`*. See getaddrinfo(3) for details.

`net.`**`EAI_*`**
All error codes *`net.getaddrinfo()`* could give. See getaddrinfo(3) for details.

`net.`**`SOL_*`**

`net.`**`IPPROTO_*`**
Levels to be used with *`net.Socket.prototype.setsockopt()`* and *`net.Socket.prototype.setsockopt()`*.

net.**SO_***

net.**IP_***

net.**IPV6_***

net.**TCP_***
Options to be used with *net.Socket.prototype.setsockopt()* and *net.Socket.prototype.setsockopt()*.

Utility functions

net.**isIP**(*address*)
Returns `4` if the given *address* is an IPv4 address, `6` if it's an IPv6 address and `0` otherwise.

net.**isIPv4**(*address*)
Returns `true` if the given *address* is a valid IPv4 address, and `false` otherwise.

net.**isIPv6**(*address*)
Returns `true` if the given *address* is a valid IPv6 address, and `false` otherwise.

os

This module exposes low level operating system facilities / syscalls.

Functions

os.**abort**()
Aborts the execution of the process and forces a core dump.

See also:

abort(3)

os.**chdir**(*path*)
Changes the current working directory of the calling process to the directory specified in *path*.

See also:

chdir(2)

os.**cloexec**(*fd*, *set*)
Sets or clears the `O_CLOEXEC` flag on the given *fd*. Since version 0.3.0 all fds are created with `O_CLOEXEC` set.

os.**close**(*fd*)
Close the given file descriptor.

See also:

close(2)

os.**dup**(*oldfd*)
Duplicates the given file descriptor, returning the lowest available one.

See also:

dup(2)

Note: Since version 0.3.0 the fds are created with `O_CLOEXEC` set. You can undo this using *os.cloexec()*.

`os.`**`dup2`**(*oldfd*, *newfd*[, *cloexec*])
Duplicates *oldfd* into *newfd*, setting the `O_CLOEXEC` flag if indicated. It defaults to `true`;

See also:

dup2(2)

`os.`**`execv`**(*path*[, *args*])

`os.`**`execvp`**(*file*[, *args*])

`os.`**`execve`**(*path*[, *args*][, *envp*])

`os.`**`execvpe`**(*file*[, *args*][, *envp*])
The `exec*` family of functions replace the current process image with a new process image.

Arguments

- **`file`** – *file* or *path* indicate file to be used for the new process image. The functions which contain a `p` in their name will search for the file in the `PATH` environment variable, or in the current directory in its absence.
- **`args`** – Arguments for the program. If an `Array` is passed, the first element should be the program filename.
- **`envp`** – Object containing the environment for the new program. The functions which do not take ab environment object will inherit it from their parent process.

See also:

execve(3)

`os.`**`exit`**([*status*])
Ends the process with the specified *status*. It defaults to 0.

See also:

exit(3)

Note: At the moment no clean shutdown is performed.

`os.`**`_exit`**(*status*)
Terminate the calling process "immediately".

See also:

_exit(2)

`os.`**`fork`**()
Creates a new process duplicating the calling process. See *os.waitpid()* for how to wait for the child process.

See also:

fork(2)

`os.`**`fstat`**(*fd*)
Same as *os.stat()* but for an already opened file descriptor.

See also:

fstat(2)

`os.`**`getegid`**`()`
Returns the effective group ID of the calling process.

See also:

getegid(2)

`os.`**`geteuid`**`()`
Returns the effective user ID of the calling process.

See also:

geteuid(2)

`os.`**`getgid`**`()`
Returns the real group ID of the calling process.

See also:

getgid(2)

`os.`**`getgroups`**`()`
Returns the supplementary group IDs of the calling process.

See also:

getgroups(2)

`os:getpid``()`
Returns the process id of the calling process.

See also:

getpid(2)

`os:getppid``()`
Returns the process id of the parent of the calling process.

See also:

getppid(2)

`os.`**`getuid`**`()`
Returns the real user ID of the calling process.

See also:

getuid(2)

`os.`**`isatty`**`(`*fd*`)`
Returns `true` if the given *fd* refers to a valid terminal type device, `false` otherwise.

See also:

isatty(3)

`os.`**`nonblock`**`(`*fd*, *set*`)`
Sets or clears the `O_NONBLOCK` flag on the given *fd*.

`os.`**`open`**`(`*path*, *flags*, *mode*`)`
Opens a file.

Arguments

- **`path`** – The file path to be opened.

- **flags** – How the file will be opened. It can be a string or an OR-ed mask of constants (listed below). Here are the supported possibilities:
  - 'r' = O_RDONLY: open the file just for reading
  - 'r+' = O_RDWR: open the file for reading and writing
  - 'w' = O_TRUNC | O_CREAT | O_WRONLY: open the file for writing only, truncating it if it exists and creating it otherwise
  - 'wx' = O_TRUNC | O_CREAT | O_WRONLY | O_EXCL: like 'w', but fails if the path exists
  - 'w+' = O_TRUNC | O_CREAT | O_RDWR: open the file for reading and writing, truncating it if it exists and creating it otherwise
  - 'wx+' = O_TRUNC | O_CREAT | O_RDWR | O_EXCL: like 'w+' but fails if the path exists
  - 'a' = O_APPEND | O_CREAT | O_WRONLY: open the file for apending, creating it if it doesn't exist
  - 'ax' = O_APPEND | O_CREAT | O_WRONLY | O_EXCL: like 'a' but fails if the path exists
  - 'a+' = O_APPEND | O_CREAT | O_RDWR: open the file for reading and apending, creating it if it doesn't exist
  - 'ax+' = O_APPEND | O_CREAT | O_RDWR | O_EXCL: like 'a+' but fails if the path exists
- **mode** – Sets the file mode (permissions and sticky bits).

Returns The opened file descriptor.

See also:

open(2)

Note: Since version 0.3.0 the fds are created with O_CLOEXEC set. You can undo this using *os.cloexec()*.

---

os.**pipe**()

Creates a *pipe* (an object that allows unidirectional data flow) and allocates a pair of file descriptors. The first descriptor connects to the read end of the pipe; the second connects to the write end. File descriptors are returned in an array: [read_fd, write_fd].

See also:

pipe(2)

Note: Since version 0.3.0 the fds are created with O_CLOEXEC set. You can undo this using *os.cloexec()*.

---

os.**read**([*nread*])

Read data from the file descriptor.

Arguments

- **nread** – Amount of data to receive. If not specified it defaults to 4096. Alternatively, a *Buffer* can be passed, and data will be read into it.

Returns The data that was read as a *Uint8Array* or the amount of data read as a number, if a *Buffer* was passed.

See also:

read(2)

`os.scandir(path)`
Lists all files in the given *path*.

See also:

scandir(3)

`os.setgid(gid)`
Sets the effective group ID of the calling process.

See also:

setgid(2)

`os.setgroups(groups)`
Sets the supplementary group IDs for the calling process.

See also:

setgroups(2)

`os.setuid(uid)`
Sets the effective user ID of the calling process.

See also:

setuid(2)

`os.setsid()`
Create a new session if the calling process is not a process group leader.

See also:

setsid(2)

`os.stat(path)`
Obtain information about the file pointed to by *path*.

Returns an object with the following properties:

- dev
- mode
- nlink
- uid
- gid
- rdev
- ino
- size
- blksize
- blocks
- flags

- gen
- atime
- mtime
- ctime
- birthtime

The `atime`, `mtime`, `ctime` and `birthtime` fields are of type Date.

See also:

stat(2)

`os.`**`ttyname`**(*fd*)
Returns the related device name of the given *fd* for which *`os.isatty()`* is `true`.

See also:

ttyname(3)

`os.`**`unlink`**(*path*)
Unlinks (usually this means completely removing) the given *path*.

See also:

unlink(3)

`os.`**`urandom`**(*bytes*)
Get *bytes* from the system CSPRNG. This is implemented by reading from `/dev/urandom`. On Linux systems supporting the getrandom(2) syscall that one is used, and in OSX arc4random_buf(3).

bytes can be an integer or a `Buffer` object. If it's an integer a `Buffer` will be returned of the specified size. If it's already a `Buffer`, if will be filled.

`os.`**`waitpid`**(*pid*[, *options*])
Wait for state changes in a child of the calling process. The return value is an object with `pid` and `status` properties. The `os.W*` family of functions can be used to get more information about the status.

See also:

waitpid(2)

`os.`**`write`**(*data*)
Write data on the file descriptor.

Arguments

- **`data`** – The data that will be written (can be a string or a *Buffer*).

Returns The number of bytes from *data* which were actually written.

See also:

write(2)

`os.`**`S_IMODE`**(*mode*)
Returns the permissions bits out of the mode field obtained with *`os.stat()`*.

`os.`**`S_ISDIR`**(*mode*)
Returns `true` if the *mode* of the file indicates it's a directory.

`os.`**`S_ISCHR`**(*mode*)
Returns `true` if the *mode* of the file indicates it's a character device.

os.**S_ISBLK**(*mode*)
Returns `true` if the *mode* of the file indicates it's a block device.

os.**S_ISREG**(*mode*)
Returns `true` if the *mode* of the file indicates it's a regular file.

os.**S_ISFIFO**(*mode*)
Returns `true` if the *mode* of the file indicates it's a FIFO.

os.**S_ISLINK**(*mode*)
Returns `true` if the *mode* of the file indicates it's a symbolic link.

os.**S_ISSOCK**(*mode*)
Returns `true` if the *mode* of the file indicates it's a socket.

os.**WIFEXITED**(*status*)

os.**WEXITSTATUS**(*status*)

os.**WIFSIGNALED**(*status*)

os.**WTERMSIG**(*status*)

os.**WIFSTOPPED**(*status*)

os.**WSTOPSIG**(*status*)

os.**WIFCONTINUED**(*status*)
Helper functions to get status information from a child process. See the man page: waitpid(2).

Constants

os.**O_***
Constants used as flags in *os.open()*.

os.**S_IF***
Flags used to check the file type in *os.stat()*.

os.**S_I***
Flags for file mode used in *os.stat()*.

os.**W***
Flags used in the options field on *os.waitpid()*.

os.**STD{IN,OUT,ERR}_FILENO**
Constants represendint default file descriptors for stdio.

path

This module provides path manipulation related functions.

Functions

path.**basename**(*p*)
Returns the last component from the pathname in *path*, deleting any trailing `/` characters.

```
path.basename('foo/bar/baz.js');
= baz.js
```

See also:

basename(3).

`path.`**`dirname`**`(`*p*`)`
The converse function of *`path.basename()`*. Returns the parent directory of the pathname in *path*.

```
path.dirname('foo/bar/baz.js');
= foo/bar
```

See also:

dirname(3).

`path.`**`join`**`(...)`
Joins the given list of partial paths into a single one using `/` as the separator.

```
path.join('foo', 'bar', 'baz');
= foo/bar/baz
```

`path.`**`normalize`**`(`*p*`)`
Normalize the given path by performing tilde expansion and then applying realpath(3). In case of error it returns the given path unchanged.

```
path.normalize('~/src/sjs');
= /Users/saghul/src/sjs
```

process

This module provides Unix process management utilities.

Process object

The *Process* object represents a child process which was spawned and provides the necessary tools to control it.

`process.Process.`**`pid`**
The process ID of the child process. Note that if `shell: true` was specified when spawning, the returned ID is that of the shell.

`class` `process.`**`Process`**`(`*options*`)`
Creates a process object. Process objects are created with the *`process.spawn()`* factory function.

`process.Process.`**`stdin`**
`io.File()` object representing the child process' *stdin*. It's write-only and any data written to it will show up as input data in the child process.

`process.Process.`**`stdout`**
`io.File()` object representing the child process' *stdout*. It's read-only and any data the child process writes to its standard output will be available to read.

`process.Process.`**`stdin`**
`io.File()` object representing the child process' *stderr*. It's read-only and any data the child process writes to its standard error will be available to read.

`process.Process.`**`wait`**
Wait untill the child process is finished. Returns an object with 2 properties:

- exit_status: the status code when the process exited. If the process didn't exit normally or due to a call to exit(3) (os exit(2)) the value will be 0.
- term_signal: the number of the signal that caused the child process to terminate. If the process didn't terminate because of a signal the value will be 0.

Functions

`process.`**`daemonize`**`()`

Detaches the current process from the terminal and continues to run in the background as a system daemon. This is perfomed using the typical Unix double-fork approach. The working directory is changed to `/` and all *stdio* file descriptors are replaced with `/dev/null`. Similar to daemon(3).

`process.`**`spawn`**`(`*cmd*`[`, *options*`])`

Creates a child process to run the given command. *cmd* should be an `Array` with the shell-escaped arguments or a string containing the full command, if the `shell` option is used.

The *options* object customizes how the child process is executed. The following properties are supported:

- *cwd*: working directory for the new process.
- *env*: object containing the environment for the new process. The calling process' environment will be used in case `null` is provided.
- *shell*: if `true` a shell will be used to spawn the command, thus running `/bin/sh -c cmd`.
- *uid*: the effective user ID for the new process.
- *gid*: the effective group ID for the new process.

Example:

```
sjs> const proc = require('process');
= undefined
sjs> var p = proc.spawn(['ls', '-l'], {stdin: null, stdout: 'pipe', stderr: null}
↪);
= undefined
sjs> print(p.stdout.read());
total 56
-rw-r--r-- 1 saghul saghul  117 May  7 23:40 AUTHORS
drwxr-xr-x 3 saghul saghul 4096 Jun  6 09:17 build
-rw-r--r-- 1 saghul saghul 3021 May 17 01:22 CHANGELOG.md
-rw-r--r-- 1 saghul saghul 3938 Jun  4 18:16 CMakeLists.txt
-rw-r--r-- 1 saghul saghul  619 May  2 11:02 CONTRIBUTING.md
drwxr-xr-x 4 saghul saghul 4096 Jun  2 23:44 docs
drwxr-xr-x 3 saghul saghul 4096 Mar 18 12:50 include
-rw-r--r-- 1 saghul saghul 1080 May  1 23:30 LICENSE
-rw-r--r-- 1 saghul saghul  667 May  6 01:38 Makefile
drwxr-xr-x 4 saghul saghul 4096 Jun  6 02:41 modules
-rw-r--r-- 1 saghul saghul 2829 Jun  3 00:52 README.md
drwxr-xr-x 4 saghul saghul 4096 Jun  2 23:42 src
drwxr-xr-x 3 saghul saghul 4096 Jun  6 09:14 test
drwxr-xr-x 2 saghul saghul 4096 Apr 22 02:16 tools

= undefined
sjs>
```

pwd

This module provides functions to query the Unix password database.

passwd object

These objects are returned by the functions in this module and represent an entry in the password database. They are a thin wrapper around a `struct passwd` structure.

Properties:

- name: user name
- passwd: user password (typically 'x', since it's encrypted)
- uid: user ID
- gid: group ID
- gecos: user information / description
- dir: user home directory
- shell: user shell

Functions

`pwd.`**`getpwnam`**`(`*name*`)`
Returns the passwd object mathing the given *name* in the Unix password database.

See also:

getpwnam(3).

`pwd.`**`getpwuid`**`(`*uid*`)`
Returns the passwd object mathing the given *uid* in the Unix password database.

See also:

getpwuid(3).

random

This module provides random number generation functions.

The current random number generator is implemented as a Mersenne Twister PRNG which is automatically seeded on startup with data from *`os.urandom()`*.

This module should be preferred to using `Math.random` since it uses a more roubust implementation. The implementation used my `Math.random` can be found here.

Note: This module could use some peer review. If you are competent to do so, please get in touch.

Random objects

class random.**Random**([*seed*])
Class implementing the Mersenne Twister PRNG. While users are free to create multiple instances of this class, the module exports the `seed` and `random` functions bound to a default instance which is seeded with the default seed.

random.Random.prototype.**seed**([*seed*])
Seed the PRNG. *seed* can be a number or an `Array` of numbers. If `null` or `undefined` is passed a default seed is obtained from *os.urandom()*. The default seed consists of 2496 bytes, enough to fill the Mersenne Twister state.

random.Random.prototype.**random**()
Return a random floating point number in the [0.0, 1.0) interval.

SystemRandom objects

class random.**SystemRandom**()
Class implementing a similar interface to *random.Random()*, but using *os.urandom()* as the source for random numbers.

random.SystemRandom.prototype.**random**()
Return a random floating point number in the [0.0, 1.0) interval.

Functions

random.**random**()
Return a random floating point number in the [0.0, 1.0) interval. The default *random.Random()* instance is used.

random.**seed**([*seed*])
Seed the default *random.Random()* instance.

system

This module provides information about the system where *sjs* is running as well as the environment. It implements CommonJS System/1.0 with some extras.

Attributes

system.**versions**
An object containing information about the *sjs* and embedded *Duktape* versions.

```
sjs> print(JSON.stringify(system.versions, null, 4))
{
    "duktape": "v1.5.0",
    "duktapeCommit": "83d5577",
    "sjs": "0.1.0"
}
```

system.**env**
Array containing the current environment variables.

`system.`**`path`**
Array containing the list of locations which are going to be used when searching for modules. It can be modified at runtime.

A list of colon separated paths can also be specified with the `SJS_PATH` variable, which will be prepended to the default paths list.

`system.`**`arch`**
System architecture (`x86`, `x64` or `arm`).

`system.`**`platform`**
String representing the running platform (`linux` or `osx`).

`system.`**`executable`**
Absolute path of the executable that started the process.

`system.`**`argv`**
Set of command line arguments that were given to the process.

```
./build/sjs t.js -i -o --verbose
// [ './build/sjs', 't.js', '-i', '-o', '--verbose' ]
```

`system.`**`args`**
Similar to `argv`, except that the interpreter binary is not listed. This is to conform with CommonJS System/1.0.

```
./build/sjs t.js -i -o --verbose
// [ 't.js', '-i', '-o', '--verbose' ]
```

`system.`**`build`**
Object providing various information about the build and the system where it was produced:

```
sjs> print(JSON.stringify(system.build, null, 4))
{
    "compiler": "GCC",
    "compilerVersion": "5.3.1",
    "system": "Linux-4.5.0-1-amd64",
    "cflags": "-pedantic -std=c99 -Wall -fstrict-aliasing -fno-omit-frame-pointer␣
↪-Wextra -O0 -g3",
    "timestamp": "2016-05-07T17:37:46Z",
    "type": "Debug"
}
```

`system.`**`endianness`**
Returns `big` if the system is Big Engian, or `little` if the system is Little Endian. This is determined at runtime.

`system.`**`stdin`**
Object of type *`io.File()`* representing the standard input.

`system.`**`stdout`**
Object of type *`io.File()`* representing the standard output.

`system.`**`stderr`**
Object of type *`io.File()`* representing the standard error.

time

This module provides time related functionality.

Functions

`time.`**`sleep`**(*secs*)
Suspends the execution of the program for the given amount of *seconds* or until a signal is received.

Note: This function may return earlier than the given amount of delay.

utils

This package contains several modules grouping utility functions.

utils.object

This object contains several utilities for working with objects. It oritinates from this module.

Functions

`utils.object.`**`format`**(*format*[, ...])
Returns a formatted string using the first argument as a `printf`-like format.

Arguments

- **`format`** – A string that contains zero or more placeholders. Each placeholder is replaced with the converted value from its corresponding argument.

Returns The formatted string.

Supported placeholders are:

- %s - String.
- %d - Number (both integer and float).
- %j - JSON. Replaced with the string '[Circular]' if the argument contains circular references.
- %% - single percent sign ('%'). This does not consume an argument.

If the placeholder does not have a corresponding argument, the placeholder is not replaced.

Examples:

```
outils.format('%s:%s', 'foo'); // 'foo:%s'
```

If there are more arguments than placeholders, the extra arguments are coerced to strings (for objects and symbols, *`utils.object.inspect()`* is used) and then concatenated, delimited by a space.

```
outils.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'
```

If the first argument is not a format string then this function returns a string that is the concatenation of all its arguments separated by spaces. Each argument is converted to a string with *`utils.object.inspect()`*.

```
outils.format(1, 2, 3); // '1 2 3'
```

`utils.object.`**`inspect`**(*object*[, *options*])
Return a string representation of *object*, which is useful for debugging.

Arguments

- **object** – The entity to be inspected.
- **options** – Optional collection of settings which control how the formatted string is constructed.

Returns A formatted string useful for debugging.

The *options* object alters certain aspects of the formatted string:

- *showHidden* - if true then the object's non-enumerable and symbol properties will be shown too. Defaults to false.
- *depth* - tells inspect how many times to recurse while formatting the object. This is useful for inspecting large complicated objects. Defaults to 2. To make it recurse indefinitely pass null.
- *colors* - if true, then the output will be styled with ANSI color codes. Defaults to false. Colors are customizable, see below.
- *customInspect* - if false, then custom inspect(depth, opts) functions defined on the objects being inspected won't be called. Defaults to true.

Example of inspecting all properties of the util object:

```
const outils = require('utils').object;
print(outils.inspect(outil, { showHidden: true, depth: null }));
```

Values may supply their own custom `inspect(depth, opts)` functions, when called they receive the current depth in the recursive inspection, as well as the options object passed to this function.

Color output (if enabled) of this function is customizable globally via `utils.object.inspect.styles` and `utils.object.inspect.colors` objects.

`utils.object.inspect.styles` is a map assigning each style a color from `utils.object.inspect.colors`. Highlighted styles and their default values are: number (yellow) boolean (yellow) string (green) date (magenta) regexp (red) null (bold) undefined (grey) special - only function at this time (cyan) * name (intentionally no styling)

Predefined color codes are: white, grey, black, blue, cyan, green, magenta, red and yellow. There are also bold, italic, underline and inverse codes.

Objects also may define their own `inspect(depth)` function which this function will invoke and use the result of when inspecting the object:

```
const outils = require('utils').object;

var obj = { name: 'nate' };
obj.inspect = function(depth) {
    return '{' + this.name + '}';
};

outils.inspect(obj);
// "{nate}"
```

You may also return another Object entirely, and the returned String will be formatted according to the returned Object. This is similar to how JSON.stringify() works:

```
var obj = { foo: 'this will not show up in the inspect() output' };
obj.inspect = function(depth) {
    return { bar: 'baz' };
```

(continues on next page)

(continued from previous page)

```
};

outils.inspect(obj);
// "{ bar: 'baz' }"
```

utils.object.**inherits**(*constructor*, *superConstructor*)
Inherit the prototype methods from one constructor into another. The prototype of *constructor* will be set to a new object created from *superConstructor*.

As an additional convenience, *superConstructor* will be accessible through the `constructor.super_` property.

utils.object.**finalizer**(*object*, *funalizerFunc*)
Set or get the finalizer for the given *object*.

Arguments

- **object** – Entity whose finalizer we are setting / getting.
- **finalizerFunc** – Function which will be called when the object is about to be freed.

Returns Undefined.

See also:

Duktape documentation on finalizers.

utils.unicode

This object provides unicode related utilities.

Functions

utils.unicode.**format**(*form*, *string*)
Normalize the given *string* to the requested unicode *form*. It uses the unorm module.

Arguments

- **form** – Type of normalizatio to be applied. One of `NFC`, `NFD`, `NFKC` or `NFKD`.
- **string** – Unicode string to be normalized.

Returns The normalized unicode string.

uuid

This module provides UUID generating and parsing functions. I was adaped from this module.

Functions

uuid.**v1**([*options*[, *buffer*[, *offset*]]])
Generate and return a RFC4122 v1 (timestamp-based) UUID.

Arguments

- **options** – Optional object with uuid state to apply. Properties may include:

  - node - (Array) Node id as Array of 6 bytes (per 4.1.6). Default: Randomly generated ID. See note 1.
  - clockseq - (Number between 0 - 0x3fff) RFC clock sequence. Default: An internally maintained clockseq is used.
  - msecs - (Number | Date) Time in milliseconds since unix Epoch. Default: The current time is used.
  - nsecs - (Number between 0-9999) additional time, in 100-nanosecond units. Ignored if msecs is unspecified. Default: internal uuid counter is used, as per 4.2.1.2.
- **buffer** – Array or buffer where UUID bytes are to be written.
- **offset** – Starting index in buffer at which to begin writing. :returns: The buffer, if specified, otherwise the string form of the UUID.

Note: The randomly generated node id is only guaranteed to stay constant for the lifetime of the current JS runtime.

---

uuid.**v4**([*options*[, *buffer*[, *offset*]]])
Generate and return a RFC4122 v4 UUID.

Arguments

- **options** – Optional object with uuid state to apply. Unused at the moment.
- **buffer** – Array or buffer where UUID bytes are to be written.
- **offset** – Starting index in buffer at which to begin writing.

Returns The buffer, if specified, otherwise the string form of the UUID.

uuid.**parse**(*id*[, *buffer*[, *offset*]])

uuid.**unparse**(*buffer*[, *offset*])
Parse and unparse UUIDs.

Arguments

- **id** – UUID(-like) string
- **buffer** – Array or buffer where UUID bytes are to be written. Default: A new Array or Buffer is used.
- **offset** – Starting index in buffer at which to begin writing. Default: 0.

2.6 FAQ

Here are some questions and answers I came up with, which potentially answer some of the ones that will eventually come up.

2.6.1 Q: WHY?!

A: The motivation for this project was mainly experimentation. If you look at today's JavaScript runtimes they seem to have a similar model: on one hand we have browsers and on the server side we have Node, but both follow a similar event-driven model. Skookum JS follows the more "traditional" model of providing all the low level primitives required to build different kinds of abstractions, an event-driven model being just one of them.

2.6.2 Q: Does it work on <insert your OS of choice>?

A: At the moment sjs works only on GNU/Linux and OSX. Adding support for other Unix platforms should be easy enough and contributions are more than welcome. Windows support is not currently planned in the short term.

2.6.3 Q: Do Nodejs modules work with Skookum JS?

A: Most likely not. Some of them might, but if a module uses any Node or browser specifics (such as `process`) it won't. In addition, sjs does not read `package.json` files nor search for modules in `node_mules` directories. This is by design.

2.6.4 Q: How can I install modules?

A: At the moment there is no packaging system, so putting them in a directory in the sjs search path is the only option. Let's say `cp` is our current package manager :-)

CHAPTER 3

Building

Compiling *sjs* is easy, the only dependency is the CMake build system.

3.1 Compile

For Debug builds (the default):

```
make
```

For Release builds:

```
make BUILDTYPE=Release
```

The installation prefix can be specified by setting `PREFIX`, it defaults to `/usr/local`.

```
make PREFIX=/usr
```

3.2 Install

sjs consists of a single binary, so if all you care about is the CLI itself, you can copy it anywhere in your filesystem. The build system provides some standard way to do this:

```
make install
```

By default *sjs* will be installed to the directory indicated by `PREFIX` (`/usr/local` by default) with the following structure:

- `PREFIX`/bin: *sjs* binary
- `PREFIX`/lib: *libsjs* library

The destination of the files can be further altered by setting `DESTDIR`. This will be prepended to `PREFIX`. Confusing, I know.

```
make DESTDIR=/tmp/testsjs install
```

3.3 Run the test suite

```
make test
```

3.4 Run the CLI without installing

```
./build/sjs
```

Index

Symbols

A

C

D

E

H

I

M

N

O

P

R

S

T

U