

# Node.js - the core

Mark Volkmann  
[mark@ociweb.com](mailto:mark@ociweb.com)  
Object Computing, Inc.  
April 21, 2012



# Overview ...

- “Node’s goal is to provide an easy way **to build scalable network programs.**”
  - <http://nodejs.org/#about>
- A **full programming environment**, not just for building “servers”
- “The **official name** of Node is “Node”.  
The unofficial name is “Node.js” to disambiguate it from other nodes.”
  - <https://github.com/joyent/node/wiki/FAQ>
- Runs on top of **Chrome V8** (see next slide)
- Implemented in **C++ and JavaScript**
- Supported on **Linux, Mac OS X and Windows**
- Created by Ryan Dahl at Joyent

passed control of the project  
to Isaac Schlueter on 1/30/12



a cartoon from substack

# ... Overview

- **Event-based** rather than thread-based; can use multiple processes
- Assumes most time consuming operations involve **I/O**
  - invoked asynchronously; non-blocking
  - a callback function is invoked when they complete



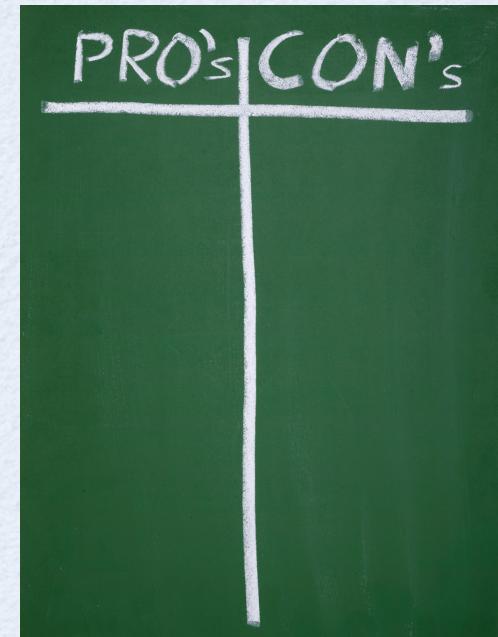
# Should You Use It?

- **Reasons To Use**

- application can benefit from asynchronous, non-blocking I/O
- application is not compute-intensive
- V8 engine is fast enough
- prefer callback or actor models of concurrency
  - over thread-based approach with synchronized access to mutable state
- same language on client and server
- like dynamically typed languages
- large number of JavaScript developers

- **Some issues being addressed**

- finding packages - there are a large number of them and finding the best ones isn't easy enough
- debugging - stack traces from asynchronously executed code are incomplete
- event loop - sometimes difficult to determine why a program isn't exiting
  - typically due to open connections



# Multiple Threads & Processes

- Node uses multiple threads internally
  - to simulate non-blocking file I/O
- You can't create new threads
  - unless you use "Threads A GoGo"
    - <https://github.com/xk/node-threads-a-gogo>
    - "provides an asynchronous, evented and/or continuation passing style API for moving blocking/longish CPU-bound tasks out of Node's event loop to **JavaScript threads that run in parallel** in the background and that use **all the available CPU cores automatically**; all **from within a single Node process**"
- Can use multiple, cooperating processes
  - see "Child Processes" core module
    - processes created with `fork` function can emit and listen for messages
  - see "Clusters" core module
    - "easily create a network of processes that all share server ports"



# Chrome V8

- From Google
- Used by Chrome browser and Node.js
- Implemented in C++
- Currently supports ECMAScript 5
- Node adopts the JavaScript syntax supported by V8
  - so will support ES6 when V8 supports it



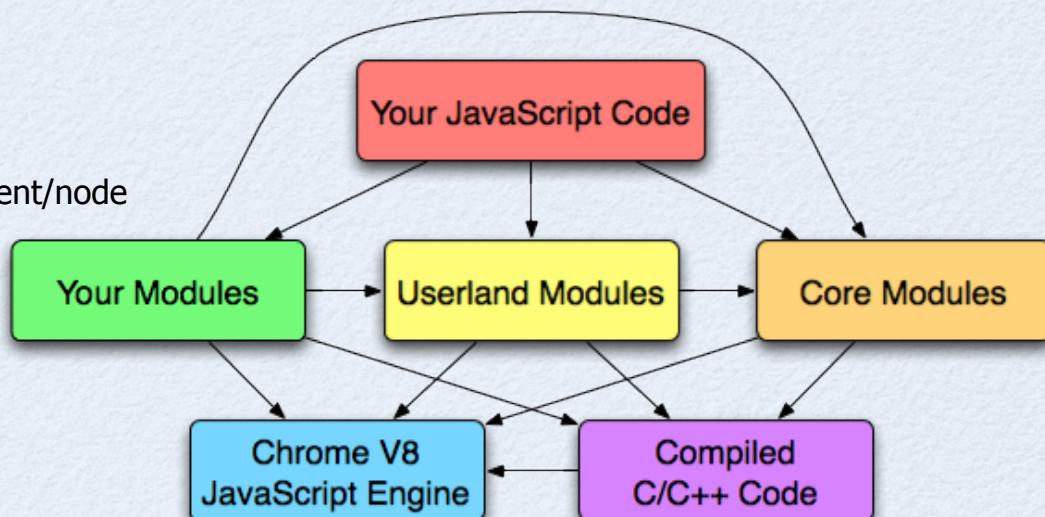
# Where To Look For Functionality

## 1. JavaScript

- core classes: `Arguments`, `Array`, `Boolean`, `Date`, `Error`,  
`Function`, `Global`, `JSON`, `Math`, `Number`, `Object`, `RegExp`, `String`

## 2. Core Modules

- included with Node
- <http://nodejs.org/docs/latest/api/>
- view source at <https://github.com/joyent/node>
  - JavaScript is in `lib` directory
  - C++ code is in `src` directory



## 3. Userland Modules (third party)

- typically installed using NPM tool
- <http://search.npmjs.org/>
- 8802 NPM packages on 4/12/12

## 4. Write yourself

Packages have JavaScript APIs,  
but can be partially implemented in C++.

# Event Loop

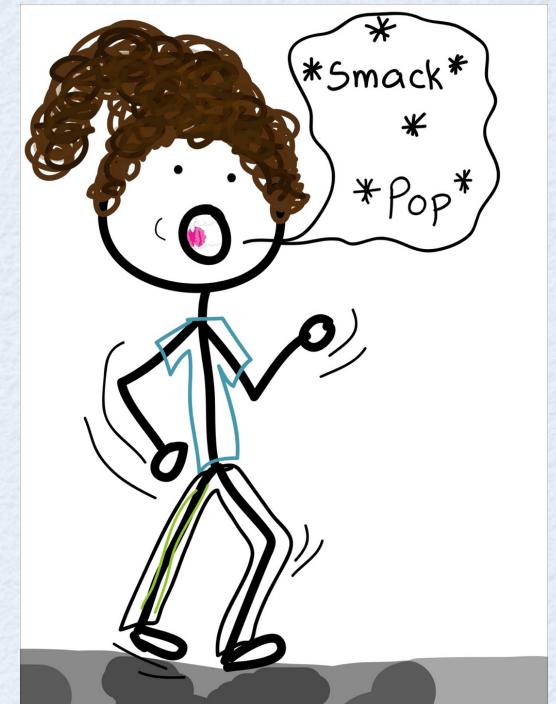
- When a Node program starts, it automatically starts an event loop
  - `node name.js`
- The currently running function, or the main script, can add function calls to the event queue
  - one way is by passing a function to `process.nextTick`
- When the currently running function completes
  - next function in event queue is removed from queue and run
- Most asynchronous functions, such as those that perform I/O
  - take a callback function as an argument
  - add a call to that function to the event queue when their work completes
- Program ends when event queue is empty
  - and there are no open network connections



U.K. roller coaster  
(1930 to 2007)

# Synchronous vs. Asynchronous

- Asynchronous functions
  - preferred over synchronous in most cases, especially when time to complete is long or unpredictable
  - take a callback function
  - invoke it, passing an error description and possibly additional arguments
- Synchronous functions
  - can make application unresponsive if long running
  - do not take a callback function
  - if an error occurs, throw an error description
    - either a string or an Error object
    - throwing an Error is preferred because when strings are thrown, no stacktrace is available



# Callbacks

- Functions passed to asynchronous functions
  - often anonymous
- Invoked any number of times
  - typically just once when operation completes
- Parameter that accepts callback
  - by convention, last parameter
  - by convention, named `cb` or `callback`
- Callback parameters
  - typically an object describing an error, if any, and a result
  - by convention, error is first argument and is named `err`
- Some libraries require following these conventions
  - ex. Async.js
- See example on next slide



# Callback Example

```
var fs = require('fs');           JavaScript

function readObject(filePath, cb) {
  fs.readFile(filePath, function (err, buf) {
    var obj = null;
    if (!err) {
      try {
        obj = JSON.parse(buf); // can throw
      } catch (e) {
        err = e;
      }
    }
    cb(err, obj);
  });
}

readObject('demo.json', function (err, obj) {
  if (err) {
    console.error(err);
  } else {
    console.log(obj);
  }
});
```

```
{                                              demo.json
  "name": "Mark Volkmann",
  "address": {
    "street": "644 Glen Summit",
    "city": "St. Charles",
    "state": "Missouri",
    "zip": 63304
  },
  "hobby": "running"
}
```

```
fs = require 'fs'           CoffeeScript

readObject = (filePath, cb) ->
  fs.readFile filePath, (err, buf) ->
    if !err
      try
        obj = JSON.parse(buf) # can throw
      catch e
        err = e
      cb err, obj

readObject 'demo.json', (err, obj) ->
  if err
    console.error err
  else
    console.log obj
```

# Async Userland Module

- Provides many functions that simplify writing asynchronous code
- Arguably the most popular Node flow control library
- Functions three categories
  - collections
    - `foreach[Series|Limit]`, `map[Series]`, `filter[Series]`, `reject[Series]`,  
`reduce[Right]`, `detect[Series]`, `sortBy`, `some`, `every`, `concat[Series]`
  - control flow
    - `series`, `parallel`, `whilst`, `until`, `waterfall`, `queue`, `auto`, `iterator`, `apply`, `nextTick`
  - utilities
    - `memoize`, `unmemoize`, `log`, `dir`, `noConflict`
- Written by Caolan McMahon
- <https://github.com/caolan/async>

# Async Example

```
var async = require('async');
var fs = require('fs');
var rimraf = require('rimraf'); // Unix "rm -rf" for Node

var dirPath = 'foo';
var fileName = 'bar.txt';
var filePath = dirPath + '/' + fileName;
var content = 'some content';
async.waterfall(
  [
    rimraf.bind(null, dirPath),
    fs.mkdir.bind(null, dirPath),
    fs.writeFile.bind(null, filePath, content),
    fs.stat.bind(null, filePath)
  ],
  function (err, stats) {
    if (err) {
      throw err;
    }
    console.log('size is ' + stats.size);
  });
});
```

# Node Globals

(other than standard JavaScript globals)

- Variables defined outside functions
  - are global in browsers
  - are local to current module in Node
- Node global variables
  - `console` - used to write to stdout and stderr
  - `global` - object that holds most global properties and functions
    - can use to share properties across modules; values can be functions
  - `process` - has methods that get info about and interact with current process; extends `EventEmitter`

more on this later
  - `require` - has property `cache` (see next slide)
  - `Buffer` - constructor function for creating objects that read and write data, especially binary data
- Node global functions
  - `require`, `setTimeout`, `clearTimeout`, `setInterval`, `clearInterval`

★

`require` is a function  
that has properties



# Node Local Variables

- Node variables that are local to current module

- **`__dirname`**

- full path to directory that contains the module source file

- **`__filename`**

- full path to source file name that defines the module

- ★ • **`module`**

- object that holds information about the current module
- shared by all instances of current module
- main property of interest is **`exports`**

- ★ • **`exports`**

- object used to export properties from a module; values can be functions
- same as `module.exports`

- **`require.cache`**

- a property on the `require` function
- holds required modules so each is only loaded once
- delete a property to allow a module to be reloaded by a subsequent call to `require`

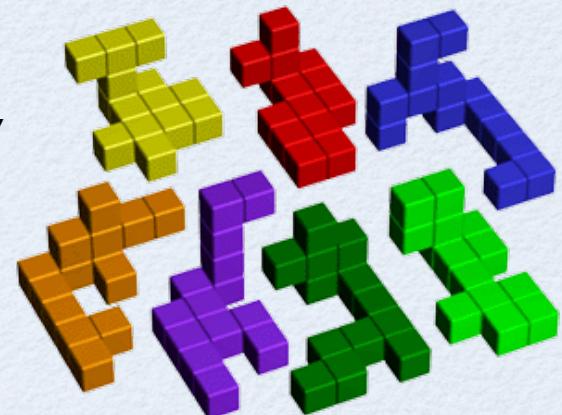
- property is full path to module, ex. `delete require.cache[__dirname + '/mymodule.js'];`



The `require` function has other properties, but they are rarely used directly. They include: `extensions`, `main`, `registerExtension` and `resolve`.

# Modules

- Defined by a single JavaScript file
  - may “require” others that are their own modules
- Top-level variables and functions defined in them are local to the module
  - not global in the entire runtime like in a browser environment
  - not visible to other modules unless exported
- Each module has its own local variable named “`module`” that refers to an object with these properties
  - ★ • `exports` - initially set to `{ };` see next slide
  - `parent` - module object of module that required this one
  - `filename` - full path to file that defines this module
  - `loaded` - false until first `require` of the module completes; defaults to `false`
  - `paths` - array of filepaths that would be searched to find this module
  - `exited` - no longer used
  - `children` - no longer used



# Defining Modules

- A module can expose functions to other modules by exporting them
  - not visible outside module if not exported
- To export many functions
  - `exports.name = some-function;`
  - repeat to export additional things
- To export a single function
  - `module.exports = some-function;`
  - replaces the default `exports` object
  - exports only one thing from the module
  - not used in conjunction with previous kind of exports
- Should also create `package.json` and `README.md`

can also export non-function values,  
including objects and arrays,  
but that isn't as common

can be a constructor function;  
can have properties whose values are other functions

used by npm

used by GitHub

A **Node package** is a collection of one or more JavaScript modules, optional C++ source files, optional shell scripts and a `package.json` file that describes the contents of the package and identifies the main module (or uses `index.js` by default).

# Using Modules

- **`var name = require('module-name');`**
  1. searches core modules
  2. searches directories listed in `NODE_PATH` environment variable
    - delimited with : in Linux and Mac or ; in Windows
  3. searches upward in path for “node\_modules” subdirectories
- **`var name = require('module-path');`**
  1. only reads from specified path; typically start with ./ or ../
- Object returned is typically
  - an object with many properties that are the exported functions
  - a constructor function
  - a single, non-constructor function
- Caches result
  - subsequent requires for same module return cached object without re-reading the file that defines the module
    - unless `require.cache` property matching full path to module is deleted

searches for specified name,  
then tries these file extensions:  
`.js, .json, .node`

for more detail, see <http://nodejs.org/docs/latest/api/modules.html>

# console Methods

similar to methods supported in browsers

- ★ • **console.log(args)** - writes to stdout with a newline
  - first arg can be a string containing formatting directives
  - if not, `util.inspect` is called on each argument (returns string representation of object)
  - formatting directives: `%s` - String, `%d` - Number, `%j` - JSON, `%%` - single percent sign
- **console.info** - same as `console.log`
- **console.warn** - same as `console.log`, but writes to stderr
- **console.error** - same as `console.warn`
- **console.dir(obj)** - writes result of `util.inspect(obj)` to stdout
- **console.time(label)** - marks start time
- **console.timeEnd(label)** - marks end time and outputs label and duration
- ★ • **console.trace** - writes stack trace to stderr
- **console.assert(boolean, msg)**
  - same as `assert.ok();` throws `AssertionError` with `msg` if false

multiple arguments  
are output with a  
space between each

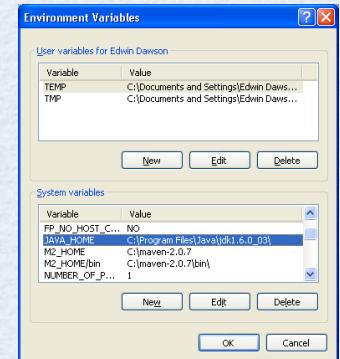


# Process Properties

- ★ • **process.argv** - array containing '`node`', main script absolute file path, and command-line arguments
- ★ • **process.env** - object containing environment variables
  - **process.pid** - process id
  - **process.stdin** - non-blocking readable stream; paused by default
    - emits '`data`' event when return key is pressed and '`end`' event when ctrl-d is pressed

see upcoming example
  - **process.stdout** and **process.stderr** - blocking, writable streams
    - important so output from asynchronous functions isn't interspersed
  - **process.title** - get/set name displayed by `ps` command; defaults to "node" 

doesn't work on Mac OS X
  - **process.version** - Node version
  - and more



# Process Methods

- `process.chdir(directory)` - changes current working directory
- `process.cwd()` - returns current working directory
- ★ • `process.exit(code)` - exits process with given status code
- `process.memoryUsage()`
  - returns object with `heapTotal` and `heapUsed` properties
- ★ • `process.nextTick(function)`
  - places given function at end of event loop queue so it runs in next iteration of event loop
  - one way to break up a long running function that avoids blocking event loop
- `process.uptime()`
  - returns number of seconds (integer, not float) process has been running
- more

see upcoming example



# Process Events

- **exit** - process is about to exit; event loop is no longer running
- **uncaughtException** - error has bubbled to top
  - if a listener is registered, uncaught exceptions will not cause a stack trace to print and program to exit
- POSIX signals - ex. SIGINT emitted when ctrl-c is pressed

```
process.on(event-name, function () {  
  ...  
});
```

for more detail on listening for events,  
see slides on **EventEmitter** later



# Buffers



- For reading and writing data, including binary data
  - some read and write functions in the “file system” module work with **Buffer** objects
- Must specify encoding when converting between strings and **Buffer** objects
  - `'ascii', 'base64', 'binary', 'hex', 'ucs2', 'utf8'`
- To create a **Buffer**
  - `new Buffer(size-in-bytes)`
  - `new Buffer(array)`
  - `new Buffer(string, encoding='utf8')`
- **Buffer** properties
  - `length` in bytes

“If you pass a **Buffer** to a function,  
it's no longer your buffer!  
Reading from it or writing to it  
at that point is entering the  
territory of undefined behavior.”  
Issac Schlueter on  
Node.js mailing list, 3/15/12

# Buffer Writing Methods ...

- **buffer[index] = value;**
  - sets a given byte
- **buffer.write(string, offset=0, length=buffer.length, encoding='utf8')**
  - length is the number of bytes to write
  - if not enough room, will write as many bytes as will fit
  - returns number of bytes written
- **buffer.writetype(value, offset, noAssert=false)**
  - where type is `Int8`, `Int16BE`, `Int16LE`, `Int32BE`, `Int32LE`,  
`UInt8`, `UInt16BE`, `UInt16LE`, `UInt32BE`, `UInt32LE`,  
`FloatBE`, `FloatLE`, `DoubleBE`, `DoubleLE`
  - when `noAssert` is `true`, it doesn't verify that there is enough space from the offset to the end of the buffer to write the type
  - no return value since the number of bytes written is known from the method name



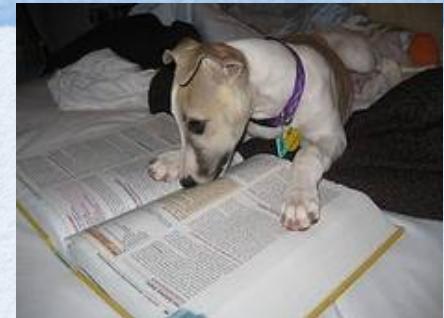
LE = Little Endian  
BE = Big Endian

# ... Buffer Writing Methods

- **buffer.copy(targetBuffer, targetStart=0, sourceStart=0, sourceEnd=buffer.length)**
  - copies data from one buffer (the method receiver) to another
- **buffer.fill(value, offset=0, end=buffer.length)**
  - **value** is used for each byte
  - **value** should be an integer (0 to 255) or a string (only first byte is used)
  - if only **value** is specified, the entire buffer is filled

# Buffer Reading Methods

- **buffer[index]**
  - returns a given byte
- **buffer.toString(encoding, start=0, end=buffer.length)**
- **buffer.readtype(offset, noAssert=false)**
  - where type is `Int8`, `Int16BE`, `Int16LE`, `Int32BE`, `Int32LE`,  
`UInt8`, `UInt16BE`, `UInt16LE`, `UInt32BE`, `UInt32LE`,  
`FloatBE`, `FloatLE`, `DoubleBE`, `DoubleLE`
  - when `noAssert` is `true`, it doesn't verify that there are enough bytes from the offset to the end of the buffer to read the type
  - returns a `Number`



LE = Little Endian  
BE = Big Endian

# Other Buffer Methods/Functions

- Other **Buffer** methods
  - **buffer.slice(start, end=buffer.length)**
    - returns a new buffer that shares memory with the receiver
    - **start** is the offset and **end** is the length of the new buffer
- **Buffer** functions
  - **Buffer.byteLength(string, encoding='utf8')**
    - returns byte length of a given string which isn't always the same as **string.length**
  - **Buffer.isBuffer(obj)**
    - determines if an object is a **Buffer**

# Buffer Example

```
var buf = new Buffer(100);

buf.writeInt16BE(12345, 0);

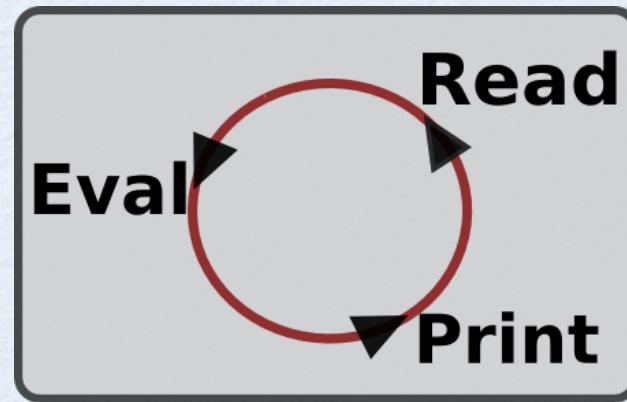
buf.writeFloatLE(Math.PI, 16);

var number = buf.readInt16BE(0);
console.log('number =', number);

var pi = buf.readFloatLE(16);
console.log('pi =', pi);
```

# REPL

- Provides a Read-Eval-Print-Loop
  - launched from a terminal window by entering "node"
  - result of last expression is held in variable `_`
- Other than entering standard JavaScript code, the following REPL commands are supported
  - `.help` - lists these commands
  - `.break` - discards a partially entered multi-line expression (ctrl-c does same)
  - `.exit` - exits REPL (ctrl-d does same)
  - `.save {file-path}` - saves every line entered in REPL to specified file
  - `.load {file-path}` - loads a JavaScript file, even if it has already been loaded; picks up changes



# Please Use A Lint Tool!

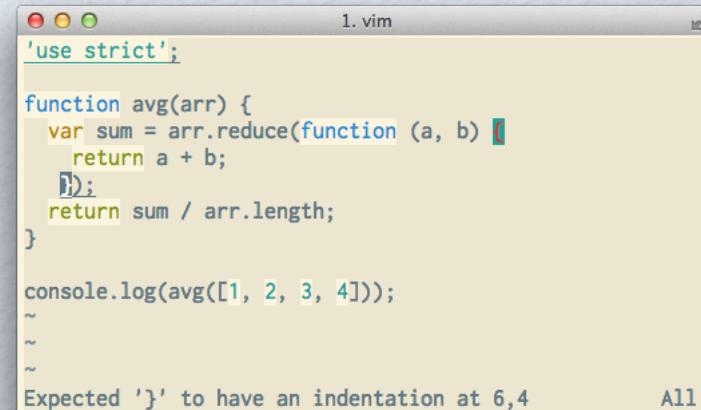
- Find coding errors and style violations, including incorrect indentation
- JSLint
  - from Douglas Crockford
  - very strict and opinionated - "Warning! JSLint will hurt your feelings."
  - <http://jslint.com/>
  - nodelint is an npm module that allows JSLint to be run from command line
    - <https://github.com/tav/nodelint>
- JSHint
  - a fork of JSLint from Anton Kovalyov, Wolfgang Kluge and Josh Perez
  - more configurable, so less opinionated
  - <http://www.jshint.com/>
  - node-jshint is an npm module that allows JSHint to be run from command line
    - <https://github.com/jshint/node-jshint>
    - `npm install -g jshint`



for more detail, see <http://nodejs.org/docs/latest/api/modules.html>

# Lint Tool Editor Integration

- Highlights errors/violations as you type!
- Emacs
  - <https://github.com/daleharvey/jshint-mode>
- Vim
  - jslint.vim - <https://github.com/hallettj/jslint.vim>
  - jshint.vim - <https://github.com/manalang/jshint.vim>
- Sublime
  - sublime-jslint - <https://github.com/fbzhong/sublime-jslint>
  - sublime-jshint - <https://github.com/uipoet/sublime-jshint>
  - Sublime Linter - <http://rondevera.github.com/jslintmate/>



```
'use strict';

function avg(arr) {
    var sum = arr.reduce(function (a, b) [
        return a + b;
    ]);
    return sum / arr.length;
}

console.log(avg([1, 2, 3, 4]));
~  
~  
~  
Expected '}' to have an indentation at 6,4
```

# Core Modules

# Overview

- Core modules are “modules and bindings that are compiled into Node”
- “In general, Node is based on the philosophy that it should not come with batteries included.”
- “One goal of Node's minimal core library is to encourage people to implement things in creative ways, without forcing their ideas onto everyone.”
- “With a tiny core and a vibrant user space, we can all flourish and experiment without the onerous burden of having to always agree”
- See links in API doc at <http://nodejs.org/docs/latest/api/>

# Table Of Contents

- Utilities - 35
- Assertion Testing - 38
- OS - 41
- Readline - 44
- TTY - 48
- Events - 50
- Path - 54
- File System - 56
- Streams - 68
- Zlib - 74
- String Decoder - 76
- Net - 77
- Datagram (UDP) - 86
- DNS - 91
- HTTP - 93
- URL - 97
- Query Strings - 100
- Crypto - 101
- TLS/SSL - 102
- HTTPS - 107
- VM - 109
- Child Processes - 113
- Cluster - 117

# Utilities ...

```
var util = require('util');
```

- **util.debug(string)** - writes to stderr preceded by "DEBUG: "
- **util.log(string)** - writes to stdout preceded by timestamp and " - "
- **util.format(fmt-string, args)**
  - returns a formatted string
  - formatting directives: %s - String, %d - Number, %j - JSON, %% - single percent sign
  - excess arguments are converted to strings using **util.inspect(arg)**
- **util.inspect(object, hidden=false, depth=2)**
  - returns string representation of an object
  - includes non-enumerable properties only if **hidden** is true
  - traverses objects to default or specified **depth**; pass **null** for infinite



# ... Utilities ...

- `util.isArray(value)` - determines if an object is an **Array**
  - in ES5, can use `Array.isArray(value)`
- `util.isDate(value)` - determines if an object is a **Date**
- `util.isError(value)` - determines if an object is an **Error**
- `util.isRegExp(value)` - determines if an object is a **RegExp**
- Use `typeof` operator for other tests
  - `typeof(value) === 'boolean' | 'number' | 'string' | 'object' | 'function' | 'undefined'`

Why not just use the `instanceof` operator in place of these?

JavaScript's `instanceof` operator doesn't work across contexts, including those created with Node's "vm" module and created in different browser windows or frames.

The `util.is*` functions provide a more reliable way to determine if an object is of one of these fundamental types.

Here's the implementation of `util.isDate`:

```
function isDate(d) {
  return typeof d === 'object' &&
    objectToString(d) === '[object Date]';
}
```

# ... Utilities

- **util.inherits(ctor, superCtor) (obj)**

- inherits prototype methods from one constructor into another
- prototype of *ctor* is set to a new object created from *superCtor*
- adds *super\_* property to *ctor*

```
var util = require('util');

function MySuper() {}

MySuper.prototype.foo = function () {
  console.log('MySuper foo entered');
};

function MySub() {
  MySuper.call(this);
}

util.inherits(MySub, MySuper);

MySub.prototype.foo = function () {
  MySub.super_.prototype.foo();
  console.log('MySub foo entered');
};

var sub = new MySub();
sub.foo();
```

can pass arguments to superclass ctor here

calls superclass method

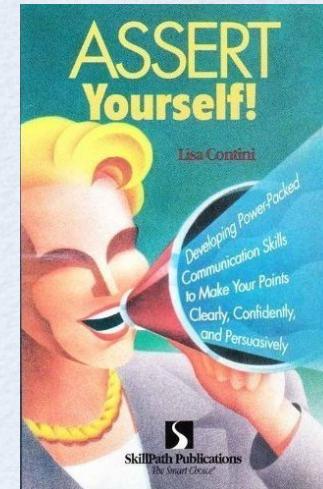
**Output:**

MySuper foo entered  
MySub foo entered

# Assertion Testing ...

```
var assert = require('assert');
```

- Basic assertions that throw an `Error` if a condition isn't met
- Used by some unit test frameworks
- Actual and expected values are specified in opposite order of many other testing APIs
- Call functions on this `assert` object that are listed on the next slide
  - ex. `assert.equal(score, 100, 'perfect score');`



# ... Assertion Testing

- Functions

- `ok(value, [message])` or `assert(value, [message])`
  - verifies that `value` is truthy; `value` can be a boolean condition
- `equal(actual, expected, [message])` - uses `==`, so performs type conversions
- `notEqual(actual, expected, [message])` - uses `!=`, so performs type conversions
- `deepEqual(actual, expected, [message])` - also compares nested properties and array elements
- `notDeepEqual(actual, expected, [message])` - ditto
- `strictEqual(actual, expected, [message])` - uses `===`, so no type conversions
- `notStrictEqual(actual, expected, [message])` - uses `!==`, so no type conversions
- `throws(fn, [error], [message])` - succeeds if `fn` throws any error or a specified one
- `doesNotThrow(fn, [error], [message])` - succeeds if `fn` does not throw any error or a specified one
- `ifError(value)` - throws if value is truthy; useful for testing first parameter in callbacks
- `fail(actual, expected, message, operator)`
  - throws `AssertionError` with `message`, ignoring other arguments
  - if `message` is null, the error message `actual + ' ' + operator + ' ' + expected` and `message` isn't used

see Node.js  
issue #2993

# Assertion Examples

```
var assert = require('assert');
var fs = require('fs');

assert(1 < 2, 'math works');

var actual = [1, [2, 3], 4];
var expected = [1, [2, 3], 4];
assert.deepEqual(actual, expected);

assert.throws(
  fs.readFileSync.bind(null, '/does/not/exist'),
  Error);

assert.doesNotThrow(
  function () {
    fs.readFileSync('demo.js');
  },
  Error);

console.log('calling fs.readFile');
fs.readFile('/does/not/exist', function (err, data) {
  assert.ifError(err);
  console.log('data =', data);
});

assert.fail(null, null, 'did not expect to be here');
```

# OS

```
var os = require('os');
```

- Retrieves information about the operating environment
  - processor architecture (ex. x64 or ia32 which are specific Intel processor architectures)
  - host name
  - load average over last 1, 5 and 15 minutes
  - OS platform (ex. 'darwin')
  - OS type (ex. 'Darwin')
  - OS release number
  - uptime in seconds
  - free and total memory in bytes
  - information about each network interface
  - information about each CPU
- Get number of processors with  
`os.cpus().length`



# OS Example

```
var os = require('os');

console.log('arch =', os.arch());
console.log('hostname =', os.hostname());
console.log('loadavg =', os.loadavg()); // 1, 5 and 15 minute load averages
console.log('platform =', os.platform());
console.log('release =', os.release());
console.log('type =', os.type());
console.log('uptime =', os.uptime(), 'seconds');

console.log('\nfreemem =', os.freemem(), 'bytes');
console.log('totalmem =', os.totalmem(), 'bytes');
var pctFree = os.freemem() / os.totalmem() * 100;
console.log('% free =', pctFree.toFixed(2) + '%');

// Returns object where keys are interface names and
// values are arrays of objects, 1 per address for the interface,
// that have address, family and internal properties.
console.log('\nnetworkInterfaces =', os.networkInterfaces());

// Returns array of objects, 1 per CPU,
// that have model, speed (in MHz) and times
// (# of CPU ticks spent in user, nice, sys, idle and irq) properties.
console.log('\ncpus =', os.cpus());
```

percentages with values between 0 and 1

**user** - milliseconds executed at user level without nice priority  
**nice** - milliseconds executed at user level with nice priority  
**system** - milliseconds executed at system level  
**idle** - milliseconds doing nothing  
**irq** - milliseconds servicing interrupts (includes waiting on I/O?)

# OS Example Output

```
arch = ia32
hostname = Mark-Volkmanns-MacBook-Pro-17.local
loadavg = [ 0.8515625, 0.67724609375, 0.64111328125 ]
platform = darwin
release = 11.2.0
type = Darwin
uptime = 15954 seconds

freemem = 3607273472 bytes
totalmem = 8589934592 bytes
```

```
networkInterfaces = { lo0:
  [ { address: 'fe80::1', family: 'IPv6', internal: true },
    { address: '127.0.0.1', family: 'IPv4', internal: true },
    { address: '::1', family: 'IPv6', internal: true } ],
  en0:
  [ { address: 'fe80::5ab0:35ff:fef3:d095', family: 'IPv6', internal: false },
    { address: '192.168.0.5', family: 'IPv4', internal: false } ],
  en1:
  [ { address: 'fe80::5ab0:35ff:fe6a:23e4', family: 'IPv6', internal: false },
    { address: '192.168.0.6', family: 'IPv4', internal: false } ] }
```

```
cpus = [ { model: 'MacBookPro6,1',
  speed: 2660,
  times: { user: 730150, nice: 0, sys: 569300, idle: 14654330, irq: 0 } },
{ model: 'MacBookPro6,1',
  speed: 2660,
  times: { user: 279490, nice: 0, sys: 126060, idle: 15548110, irq: 0 } },
{ model: 'MacBookPro6,1',
  speed: 2660,
  times: { user: 985960, nice: 0, sys: 616860, idle: 14350840, irq: 0 } },
{ model: 'MacBookPro6,1',
  speed: 2660,
  times: { user: 254950, nice: 0, sys: 115040, idle: 15583670, irq: 0 } } ]
```

lo0 is the “virtual loopback interface”.  
Packets sent to this are routed **internally** to the network loopback.  
This bypasses local network interface hardware, reducing load on network resources.

# Readline ...

```
var rl = require('readline');
```

- Reads streams one line at a time
- Supports many control keys when reading from stdin
  - see `_twrite` method in `readline.js`
- Functions
  - `createInterface(input, output, completer)`
    - `input` and `output` are streams
      - typically `process.stdin` and `process.stdout` (should probably be the defaults)
      - can be a file streams
    - `completer` is a function that provides tab completion; can omit; see example ahead
  - `setPrompt(prompt, [length])`
    - sets prompt displayed when `prompt()` is called
    - if `length` is specified, the prompt will be right-padded with spaces to meet it
  - `prompt()`
    - outputs prompt specified in `setPrompt()`
    - user can enter a line without a prompt

Key	Action
ctrl-c	emits SIGINT
ctrl-z	emits SIGTSTP
ctrl-b	back one char
ctrl-f	forward one char
ctrl-left	go to left word boundary
ctrl-right	got to right word boundar
ctrl-a	go to start
ctrl-e	go to end
ctrl-h	delete char to left of cursor
ctrl-d	delete char under cursor
ctrl-u	delete entire line
ctrl-k	delete to end
ctrl-w	delete back to word boundary
ctrl-backspace	same as ctrl-w
ctrl-delete	delete forward to word boundary
ctrl-p	previous in history
ctrl-n	next in history

# ... Readline

## More Functions

- **question(text, callback)**
  - outputs `text` instead of the specified prompt
  - passes line entered by user to callback
  - does not emit '`line`' event, but if additional lines are entered, '`line`' events are generated for those
- **write(text)** - writes string to input as if user typed it
- **pause()** - used internally for tab completion
- **resume()** - used internally for tab completion
- **close()** - marks interface as closed and emits '`close`' event, but doesn't close input stream

## Events

- '`line`' - when user presses enter key or there is a newline in the stream
- '`close`' - when `close()` is called or ctrl-c or ctrl-d are pressed

```
var rl = require('readline');

var intf = rl.createInterface(
  process.stdin, process.stdout);

intf.on('line', function (line) {
  // Only invoked if more than one line is entered.
  // The question method doesn't emit this event.
  console.log('line event: got ' + line);
});

intf.question('What is your name? ',
  function (name) {
    console.log('Hello, ' + name + '!');
  });
}
```

# Readline Guess Example

```
var rl = require('readline');

// Generate a random number between 1 and 10.
var answer = Math.floor(Math.random() * 10) + 1;
console.log('A number between 1 and 10 has been selected.');

var intf = rl.createInterface(process.stdin, process.stdout);
function prompt(msg) {
    intf.setPrompt(msg);
    intf.prompt();
}

intf.on('line', function (line) {
    var number = parseInt(line, 10);
    if (isNaN(number)) {
        prompt('Enter a number: ');
    } else if (number < answer) {
        prompt('Too low: ');
    } else if (number > answer) {
        prompt('Too high: ');
    } else {
        console.log('CORRECT!');
    }
    // Allow the program to terminate.
    intf.close();
    process.stdin.destroy();
})
);

prompt('Guess the number: '');
```

# Readline Completion Example

```
var rl = require('readline');
var fruits = ('apple banana blackberry blueberry cherry grape grapefruit ' +
    'lemon lime orange peach pear plum strawberry').split(' ');

function completer(partial) {
    var options = fruits.filter(function (word) {
        return word.indexOf(partial) === 0;
    });
    return [options, partial];
}

console.log('Enter names of fruits.');
console.log('Press tab for completion.');
console.log('To exit, enter "exit" or press ctrl-c or ctrl-d.');
var intf = rl.createInterface(process.stdin, process.stdout, completer);
intf.setPrompt('fruit: ');
intf.prompt();
intf.on('line', function (line) {
    if (line === 'exit') {
        intf.close();
        process.stdin.destroy(); // allows program to terminate
    } else {
        console.log('got ' + line);
        intf.prompt();
    }
});
```

gets fruits that start with entered text

```
// Asynchronous version
function completer(partial, cb) {
    var options = fruits.filter(function (word) {
        return word.indexOf(partial) === 0;
    });
    cb(null, [options, partial]);
}
```

# TTY

```
var tty = require('tty');
```

- Intercepts terminal keystrokes
  - including whether shift, ctrl and meta keys were down
- Important for intercepting certain keystrokes before the operating system acts on them
  - for example, ctrl-c normally sends an interrupt signal (SIGINT) that causes a Node program to stop



# TTY Example

```
var tty = require('tty');

process.stdin.resume(); // must do before entering raw mode
tty.setRawMode(true); // can't intercept key presses without this

// char string is only set for normal characters.
// key object is set for all key presses.
// Properties of key include name, ctrl, meta and shift.
process.stdin.on('keypress', function (char, key) {
    console.log('char =', char);
    if (key) {
        console.log('key =', key);
        var name = '';
        if (key.shift) key.name = key.name.toUpperCase();
        if (key.meta) name += 'meta ';
        if (key.ctrl) name += 'ctrl ';
        name += key.name;
        console.log('You pressed ' + name);
        if (key.ctrl && key.name == 'c') {
            console.log('exiting');
            process.exit()
        }
    }
});
```

On Mac OS X, neither the option nor the command key cause the `meta` property to be set!

# Events

- Many Node classes inherit from `EventEmitter`
- Custom classes can also
- Objects that are event emitters
  - always emit '`newListener`' when listeners are added
  - often emit '`error`' when an error occurs in one of their methods
- Event listeners
  - functions that are invoked when events are emitted
  - passed any data emitted with the event
  - not passed the event name unless `EventEmitter` subclasses are specifically written to do so

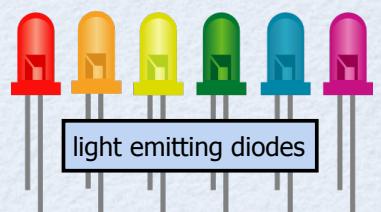


# EventEmitter Methods ...

```
var EventEmitter = require('events').EventEmitter;
```

only property exported

- **setMaxListeners(n)**
  - sets the maximum number of listeners that can be registered for a given event
  - default is 10; set to zero for unlimited
  - useful for finding bugs where an excessive number listeners are being registered
  - outputs warning using `console.error` and calls `console.trace`, but does not throw
- **on(event, listener) Or addListener(event, listener)**
  - registers a listener function for a given event
- **once(event, listener)**
  - registers a listener function for a given event and removes it after its first invocation
- **emit(event, args)**
  - invokes listener functions for the event in the order they registered; passes all args to them
  - synchronous! - listener functions are run immediately, not added to event loop queue
  - workaround
    - listener functions can add a function to event loop queue by passing it to `process.nextTick`



# ... EventEmitter Methods

- **removeListener(event, listener)**
  - unregisters a listener function for a given event
- **removeAllListeners([event])**
  - unregisters all listener functions for a given event or all events
- **listeners(event)**
  - returns a live array of all listener functions for a given event
  - can delete function elements to unregister
  - can push function elements to register

# Event Example

```
var EventEmitter = require('events').EventEmitter;
var util = require('util');

function Thermostat() {}
util.inherits(Thermostat, EventEmitter);

Thermostat.prototype.set = function (temperature) {
  this.temperature = temperature;
  if (temperature < 32) {
    this.emit('cold', temperature);
  } else if (temperature > 100) {
    this.emit('hot', temperature);
  }
};

var t = new Thermostat();
t.on('cold', function (temp) {
  console.log(temp + ' is too cold!');
});
t.on('hot', function (temp) {
  console.log(temp + ' is too hot!');
});

t.set(50);
t.set(0); // outputs "0 is too cold!"
t.set(110); // outputs "110 is too hot!"
```

# Path

```
var path = require('path');
```

- Methods

*p* parameters are string file paths

- **normalize(*p*)** - returns a new path after resolving .. and ., and replacing consecutive slashes with one, in path *p*
- **join(*path-parts*)** - returns a path created by joining any number of path parts and normalizing
- **resolve([*from...*], *to*)** - resolves relative path *to* to an absolute path also normalizes by prefixing with the *from* values from right to left, why right to left? using the first combination found to exist or the current directory if none exist
- **relative(*from*, *to*)** - returns a relative path that can be used to cd from *from* to *to*
- **dirname(*p*)** - returns the directory portion of path *p*
- **basename(*p*, [*ext*])** - returns the filename portion of path *p*, removing the extension *ext* if supplied
- **extname(*p*)** - returns the extension of path *p*, including a leading dot
- **exists(*p*, *callback*)** - passes a boolean to callback indicating whether the path *p* exists
- **existsSync(*p*)** - returns a boolean indicating whether the path *p* exists



moved to the "File System" module in Node version 7

# Path Examples

```
var path = require('path');

console.log(path.normalize('../fs/../console///demo.js'));
// ../console/demo.js

var dirs = ['../url', '../vm', '../zlib'];
var args = dirs.concat('demo.js');
console.log(path.resolve.apply(null, args));
// /Users/Mark/Documents/OCI/SVN/training/Node.js/labs/zlib/demo.js

var absPath = path.resolve('../foo.txt');
// Recall that __dirname holds the absolute path to the current directory.
// var absPath = __dirname + '/foo.txt'; // same as above

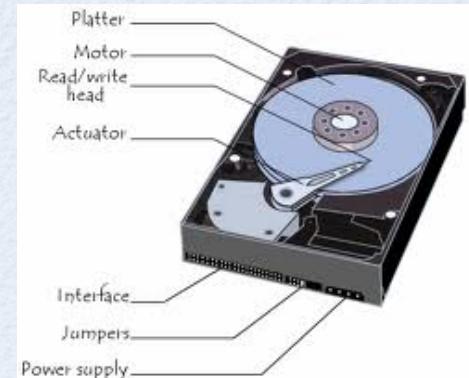
console.log(path.dirname(absPath)); // parent of current directory
console.log(path.basename(absPath, '.txt')); // foo
console.log(path.extname(absPath)); // .txt

path.exists(absPath, function (existsP) {
  console.log(absPath + ' exists? ' + existsP); // false
});
```

# File System

```
var fs = require('fs');
```

- Wraps access to POSIX file I/O functions
- Provides asynchronous (preferred) and synchronous versions of most functions
  - asynchronous functions take a callback function as their last argument
    - callback functions take an error description as their first argument
  - synchronous functions can throw errors
- Contains many more functions than any other core module
  - buckle up, six slides worth coming next!
  - for parameter details, see <http://nodejs.org/docs/latest/api/fs.html>



# File System Functions ...

- Open/Close
  - `open/openSync` - takes a path and returns a file descriptor
  - `close/closeSync` - takes a file descriptor
- Reading
  - `read/readSync` - takes a file descriptor and a `Buffer`; reads specified range of bytes from file into `Buffer`
  - `readFile/readFileSync` - takes a file path; reads entire file; returns data in a `Buffer`
- Writing
  - `write/writeSync` - takes a file descriptor and a `Buffer`; writes specified range of bytes from `Buffer` into file starting at a given position
  - `writeFile/writeFileSync` - takes a file path and a string or `Buffer`; writes bytes in string or `Buffer` to file, replacing existing content



Use of async functions instead of sync functions is strongly encouraged to avoid blocking the event loop with long-running I/O operations.

- `'r'` - Open file for reading. An exception occurs if the file does not exist.
- `'r+'` - Open file for reading and writing. An exception occurs if the file does not exist.
- `'w'` - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'w+'` - Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'a'` - Open file for appending. The file is created if it does not exist.
- `'a+'` - Open file for reading and appending. The file is created if it does not exist.

modes are used by these functions and their "Sync" counterparts:  
`chmod`, `fchmod`, `lchmod`,  
`mkdir` and `open`

# ... File System Functions ...

- Streams

- `createReadStream` - returns an `fs.ReadStream` object
- `createWriteStream` - returns an `fs.WriteStream` object
- see detail on stream objects later

- Directories

- `mkdir/mkdirSync` - takes file path and optional access permissions mode (ex. '755') and creates a directory
- `readdir/readdirSync` - takes file path and gets array of directory contents
- `rmdir/rmdirSync` - takes directory path and deletes directory only if empty

- Links

- `link/linkSync` - creates a file that is a link (a.k.a. hard link) to another
- `symlink/symlinkSync` - creates a file that is a symbolic link (a.k.a. soft link) to another
- `readLink/readLinkSync` - gets info. about the file referred to by a link
- `unlink/unlinkSync` - deletes a link or file; note there is no `rm` function



For more on **hard links**, see  
[http://en.wikipedia.org/wiki/Hard\\_link](http://en.wikipedia.org/wiki/Hard_link)

For more on **symbolic links**, see  
[http://en.wikipedia.org/wiki/Symbolic\\_link](http://en.wikipedia.org/wiki/Symbolic_link)

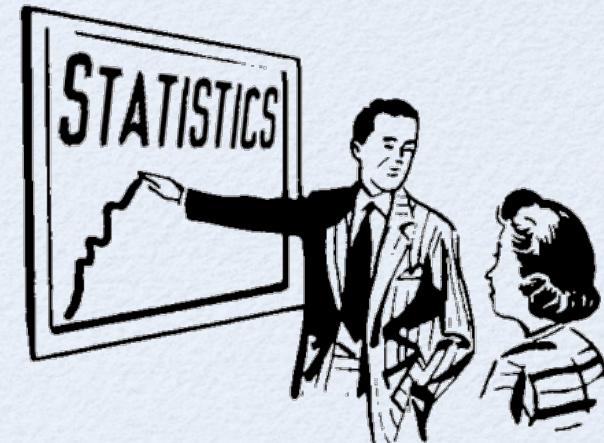
# ... File System Functions ...

## ● Statistics

- **stat/statSync** - takes a file path; returns an **fs.Stats** object that provides details about the file
- **fstat/fstatSync** - same as **stat** versions, but takes a file descriptor object instead of a file path
- **lstat/lstatSync** - same as **stat** versions, but if file path is to a link, describes the link instead of the target file
- all return an **fs.Stats** object
  - methods: **isFile**, **isDirectory**, **isBlockDevice**, **isCharacterDevice**, **isSymbolicLink**, **isFIFO**, **isSocket**
  - properties: **dev**, **ino**, **mode**, **nlink**, **uid**, **gid**, **rdev**, **size**, **blksize**, **blocks**, **atime**, **mtime**, **ctime**
  - **atime**, **mtime** and **ctime** are **Date** objects

## ● Timestamps

- **utimes/utimesSync** - takes a file path, atime and mtime; changes atime and mtime values for the file
- **futimes/futimesSync** - same as **utimes** versions, but takes a file descriptor object instead of a file path



**atime** is time of last access

**mtime** is time of last content modification

**ctime** is time of last content, owner or permission change

# ... File System Functions ...

- Change owner
  - **chown/chownSync** - takes a file path, user id and group id
  - **fchown/fchownSync** - same as **chown** versions, but takes a file descriptor instead of a file path
  - **lchown/lchownSync** - same as **chown** versions, but if file path is to a link, changes the link instead of the target file
- Change mode (access permissions)
  - **chmod/chmodSync** - takes a file path and a mode (an octal number or a string) ex. 0755 or '755'
    - octal literals are not allowed in ES5 strict mode
  - **fchmod/fchmodSync** - same as **chmod** versions, but takes a file descriptor instead of a file path
  - **lchmod/lchmodSync** - same as **chmod** versions, but if file path is to a link, changes the link instead of the target file



# ... File System Functions ...



## • Watching

- **watchFile** - takes a file path, optional options and a callback
  - file path cannot be to a directory
  - calls callback every time the file is accessed (not under Mac OS X!) or modified
  - default options are typically good; see doc for detail
  - callback is passed current and previous `fs.Stats` objects
  - to detect file modification, compare current mtime value to previous one
- **unwatchFile** - takes a file path; stops watching for file access
- **watch** - takes a file path, optional options and a callback
  - file path can be to a directory (typical case)
  - does not detect changes in nested directories
  - default options are typically good; see doc for detail
  - callback is invoked when the file or directory being watched has a change
    - passed an event string (always 'change') and the associated file path (useful when watching a directory and a file in it changes)
  - returns an `fs.FSWatcher` object that emits 'change' and 'error' events and has a `close` method

Currently the file path isn't consistently passed to the callback. It never is under Mac OS X because that OS doesn't provide the information.

When `watch` indicates that the content of a directory has changed, `fs.readdir` can be used to determine which files have changed. See the `watch` example coming up.

**From Ben Noordhuis** ... " `fs.watch` on OS X and the BSDs is backed by the `kqueue` event mechanism. It has a couple of known shortcomings, lack of \*time updates being one of them.

Linux and Windows use the `inotify` and `ReadDirectoryChangesW` interfaces respectively, which are more robust.

Use `fs.watchFile` if you want consistent behavior across Unices. It's not nearly as efficient as `fs.watch` though, and it isn't supported on Windows."

# ... File System Functions

- Other

- **fsync/fsyncSync** - synchronizes in-memory data with data on disk
- **realpath.realpathSync** - resolves relative file paths to absolute paths
- **rename/renameSync** - renames and/or moves a file
  - takes "from path" and "to path"; "to path" must include file name, not just directory
- **truncate/truncateSync** - truncates or extends a file to a given byte length

# Reading Files - Three Ways

```
var async = require('async');
var fs = require('fs');
var filePath = 'foo.txt';

fs.readFile(filePath, function (err, buf) {
  if (err) throw err;
  console.log(buf.toString());
});

var rs = fs.createReadStream(filePath);
rs.on('data', function (buf) {
  console.log(buf.toString()); can be called multiple times
}); for large files
  can also listen for 'error' events

var maxSize = 100;
var buf = new Buffer(maxSize); ←
var openFile = fs.open.bind(null, filePath, 'r');
var readFile = function (fd, cb) {
  fs.read(fd, buf, 0, buf.length, 0, function (err, bytesRead) {
    console.log(buf.toString());
    cb(err, fd);
  });
};
async.waterfall([openFile, readFile, fs.close], function (err) {
  if (err) throw err;
}); see more on async module
in "Userland Modules" section
```

This approach has the following advantages:  
• can read from a specified chunk of the file  
• can read into a specified chunk of the **Buffer**  
Disadvantages include:  
• all the things above MUST be specified  
• the code is longer and more complicated

# Writing Files - Three Ways

```
var async = require('async');
var fs = require('fs');

var filePath = 'foo.txt';
var data = 'red\ngreen\nblue\n';

// If file already exists, content is replaced.
fs.writeFile(filePath, data, function (err) {
  if (err) throw err;
});

var ws = fs.createWriteStream(filePath);
ws.write(data);      can listen for 'error' events
ws.end();

var buf = new Buffer(data);
var open = fs.open.bind(null, filePath, 'w');
var write = function (fd, cb) {
  fs.write(fd, buf, 0, buf.length, 0, function (err) {
    cb(err, fd);
  });
}
async.waterfall([open, write, fs.close], function (err) {
  if (err) throw err;
});
```

This approach has the following advantages:

- can write into a specified chunk of the file
- can write from a specified chunk of the `Buffer`

Disadvantages include:

- all the things above MUST be specified
- the code is longer and more complicated

# Watch Example ...

```
var async = require('async');
var fs = require('fs');
var dir = '.';

function getStats(dir, cb) {
  fs.readdir(dir, function (err, files) {
    if (err) return cb(err);

    var stats = {};

    // This function is passed to async.every below.
    var iterator = function (file, cb) {
      // Skip hidden files (start with a period).
      if (/^\./.test(file)) return cb(true);
      // Skip Vim backup files (end with a tilde).
      if (/~$/.test(file)) return cb(true);

      fs.stat(file, function (statErr, stat) {
        if (statErr) {
          err = statErr;
        } else {
          stats[file] = stat;
        }
        cb(!err); // stops async.every when there is an error
      });
    };
    async.every(files, iterator, function (result) {
      cb(err, stats);
    });
  });
}

// Gets an fs.Stats object for every file in a given directory.
// dir is a directory path.
// cb is a callback that is passed err and an array of fs.Stats objects.
```

# ... Watch Example ...

```
function report(name, oldStat, newStat) {
  if (!oldStat && newStat) {
    console.log(name, 'was created');
    return;
  }

  var modified = newStat.mtime > oldStat.mtime;
  if (modified) {
    var diff = newStat.size - oldStat.size;
    var suffix = Math.abs(diff) === 1 ? 'byte' : 'bytes';
    var desc =
      diff > 0 ? 'increased by ' + diff + ' ' + suffix :
      diff < 0 ? 'decreased by ' + -diff + ' ' + suffix :
      'did not change';
    console.log(name, 'content modified, size', desc);
  }
}

var oldStats;
getStats(dir, function (err, stats) {
  oldStats = stats;
});
```

Reports activity for a single file.  
name is a file name.  
oldStat and newStat are fs.Stats objects.

# ... Watch Example

```
fs.watch(dir, function () {
  getStats(dir, function (err, newStats) {
    if (err) {
      return console.error(err);
    }

    Object.keys(oldStats).forEach(function (name) {
      if (!newStats[name]) {
        console.log(name, 'was deleted');
      }
    });

    Object.keys(newStats).forEach(function (name) {
      report(name, oldStats[name], newStats[name]);
    });

    oldStats = newStats;
  });
});
```

Under Mac OS X, `null` is always passed to the callback for `filePath`.

The callback is invoked when any file in the directory is created, deleted, or has its contents modified.

It is not invoked when

- a file is merely accessed
- the owner of a file is changed
- the permissions on a file are changed

# Streams

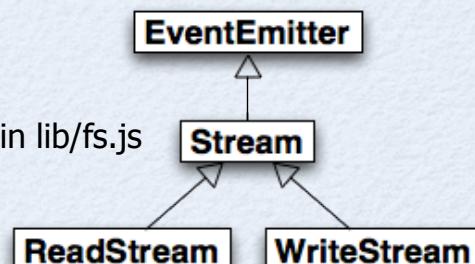
- Two types
  - readable streams - created by `fs.createReadStream(file-path, [options])`
  - writable streams - created by `fs.createWriteStream(file-path, [options])`
  - options include `flags` (a mode at bottom of slide 11) and `encoding ('ascii', 'base64', 'binary', 'hex', 'ucs2' or 'utf8')`
- A stream can be one or both (duplex) types
- Classes
  - `Stream` inherits from `EventEmitter` defined in lib/stream.js
  - `ReadStream` and `WriteStream` inherit from `Stream` defined in lib/fs.js



There are several ways to create duplex streams in the core modules including:  
the `Stream` pipe method,  
the `net.createServer` and `net.connect` functions  
(both return a `net.Socket` object which is a duplex stream)  
and the `tls.connect` function.

Examples of non-duplex streams include:  
`http.ServerRequest`, `http.ServerResponse`,  
`fs.ReadStream` and `fs.WriteStream`.

Custom streams of both types can also be created.  
For an example, see <https://github.com/dominictarr/event-stream>.



# Readable Streams ...

- Events
  - **open** - when stream is ready; callback is passed a file descriptor object
  - **data** - when data has been read
    - callback is passed a `Buffer` object or a string if `setEncoding` was called on the stream
  - **end** - when end of stream is reached
    - no more '`data`' events will be emitted
  - **error** - when a read error occurs
  - **close** - when underlying file descriptor is closed
- Properties
  - **readable** - boolean indication of whether the stream can be read
    - changes to false if an `error` or `end` event is delivered or the `destroy` method is called on the stream

# ... Readable Streams

- Methods

- `setEncoding(encoding)` - sets character encoding used
  - valid values are `'ascii'`, `'base64'`, `'binary'`, `'hex'`, `'ucs2'` and `'utf8'`
- `pause()` - temporarily stops `'data'` events
- `resume()` - resumes `'data'` events
- `destroy()` - closes underlying file descriptor
  - no more events will be emitted after `close`
- `destroySoon()` - closes underlying file descriptor
  - only after writes complete if the stream is also writable
- `pipe(destination, [options])` - connects this stream to a writable stream

- See example on slide 17

```
var fs = require('fs');
var rs = fs.createReadStream('TaleOfTwoCities.txt');
rs.pipe(process.stdout);
```

everything read from the file  
is written to stdout

# Reading a File By Lines

```
var fs = require('fs');

function readLines(filePath, cb) {
  var rs = fs.createReadStream(filePath, {bufferSize: 80});
  var leftover = '';

  rs.on('data', function (buf) {
    var lines = buf.toString().split('\n');
    lines[0] = leftover + lines[0];
    leftover = lines.pop(); // chunk at end
    lines.forEach(function (line) {
      cb(line);
    });
  });

  rs.on('end', function () {
    if (leftover.length > 0) {
      cb(leftover);
    }
  });
}

readLines('./story.txt', console.log);
```

callback is invoked once for each line

See slightly better implementation in node-liner userland module.  
npm install liner

# Writable Streams ...

- Events

- **open** - when stream is ready; callback is passed a file descriptor object
- **drain** - when “kernel buffer” is empty meaning it is safe to write again
- **error** - when write error occurs
- **close** - when underlying file descriptor has been closed
- **pipe** - when stream is passed to **pipe** method of a **ReadStream**

Kernel buffers are used internally by Node to buffer output in case destination streams cannot keep up.

- Properties

- **bytesWritten** - number of bytes written so far
- **writable** - boolean indication of whether stream can be written
  - changes to **false** if an **error** event is delivered or the **end** or **destroy** method is called on the stream (see next slide)

# ... Writable Streams

- Methods

- `write(string, encoding='utf8', [fd])` - writes a string to stream
  - `fd` parameter is a UNIX-only, rarely used option
- `write(buffer)` - writes contents of `Buffer` to stream

---
- `end()` - terminates stream
- `end(string, [encoding])` - writes a string to stream and then terminates it
- `end(buffer)` - writes contents of `Buffer` to stream and then terminates it

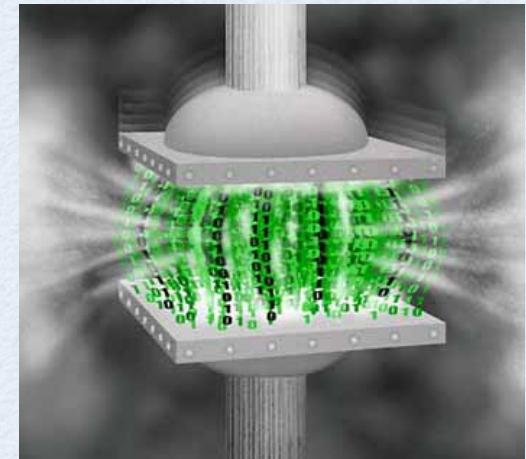
---
- `destroy()` - closes underlying file descriptor
  - no more events will be emitted after `close`
- `destroySoon()` - closes underlying file descriptor
  - only after writes complete if stream is also writable

- See example on slide 18

# Zlib

```
var zlib = require('zlib');
```

- Supports three kinds of compression and decompression
  - Deflate - from Wikipedia, "Deflate is a lossless data compression algorithm that uses a combination of the LZ77 algorithm and Huffman coding."
  - Deflate Raw - same as Deflate, but doesn't append a zlib header
  - GZIP - based on the Deflate algorithm
- Highly configurable
- Seven classes - create instances with `zlib.createName([options])` ;
  - `Gzip`, `Deflate` and `DeflateRaw` are writeable streams that compress
  - `Gunzip`, `Inflate` and `InflateRaw` are readable streams that decompress
  - `Unzip` is a readable stream that detects the compression type and decompresses
- Convenience functions
  - perform seven operations corresponding to the seven classes without streams
    - `deflate`, `deflateRaw`, `gzip`, `gunzip`, `inflate`, `inflateRaw` and `unzip`
  - each takes a string or `Buffer` object and a callback function that is passed an `Error`, if any, and the result as a `Buffer`



# Zlib Example

```
var fs = require('fs');
var zlib = require('zlib');

function zipToFile(data, filePath, cb) {
    zlib.gzip(data, function (err, buffer) {
        if (err) return cb(err);
        fs.writeFile(filePath, buffer, cb);
    });
}

function unzipFromFile(filePath, cb) {
    fs.readFile(filePath, function (err, buffer) {
        if (err) return cb(err);
        zlib.gunzip(buffer, function (err, buffer) {
            cb(err, buffer.toString());
        });
    });
}

var filePath = 'message.gz';
var data = 'This is a message';
zipToFile(data, filePath, function (err) {
    if (err) throw err;
    unzipFromFile(filePath, function (err, result) {
        if (err) throw err;
        console.log('result =', result);
    });
});
});
```

# String Decoder

```
var StringDecoder = require('string_decoder').StringDecoder;
```

- Not documented yet
- Handles writing data from buffers that do not end in a complete multi-byte character
- Used by
  - core modules `fs`, `http`, `net`, `repl` and `tls`
  - npm's `read` module
    - which it uses for "`npm init`" to prompt for `package.json` information



# Net ...

```
var net = require('net');
```

- Provides methods for implementing TCP servers and clients
- Methods
  - **createServer([options], [callback])**
    - typically used server-side
    - returns a `net.Server` object
    - callback is passed a `net.Socket` object
    - register listeners for events on socket in callback
  - **connect(port, [host], [callback])** - for TCP  
**connect(path, [callback])** - for Unix socket
    - asynchronously creates a new connection
    - typically used client-side
    - returns a `net.Socket` object and passes nothing to callback
    - `host` defaults to localhost
  - **createConnection(args)**
    - alias for `connect` method



for communicating between processes on same host

# ... Net

- Methods

- **isIP(s)** - returns 0 if **s** is not an IP address string, 4 if IPv4, and 6 if IPv6
- **isIPv4(s)** - returns boolean indicating whether **s** is a version 4 IP address string
  - pattern is **d.d.d.d** where **d** is an integer between 0 and 255
  - can be represented in 32 bits
- **isIPv6(s)** - returns boolean indicating whether **s** is a version 6 IP address string
  - pattern is **h:h:h:h:h:h:h:h** where each **h** is a 4 character hex value
  - can be represented in 128 bits
  - leading zeros in an **h** value may be omitted
  - **h** values that are all zeros can be replaced by a single zero or omitted
  - all colons must be retained, except more than two consecutive colons can be replaced by only two colons once within an address
    - ex. **1:2:0:0:0:0:7:8** is equivalent to **1:2:::::7:8** and **1:2::7:8**

# net.Server Class ...

- Kind of object returned by `net.createServer` function
- Methods
  - `listen(port, [host], callback)` - for TCP  
`listen(path, callback)` - for Unix socket
    - listens for new connections
    - if `host` is omitted, will listen for connections from any host
    - returns nothing and passes nothing to callback
  - `pause(ms)`
    - stop accepting new connections for `ms` milliseconds, perhaps for throttling
  - `close()`
    - asynchronously stop accepting new connections permanently
    - a 'close' event is emitted when complete
  - `address()`
    - returns an object containing `port` and `address` (IP) properties



# ... net.Server Class

- Events

```
register for these with  
server.on(event-name, callback);
```

- **listening** - emitted when server is ready to accept connections
- **connection** - emitted when a connection is made
  - `net.Socket` object is passed to callback
- **close** - emitted when server is no longer accepting connections
- **error** - emitted when an error occurs
  - `Error` object is passed to callback

- Properties

- **maxConnections** - set to limit number of connections
- **connections** - will be set to current number of connections



# net.Socket Class ...

- Represents a TCP or Unix socket
- Kind of object returned by `net.connect` function
- Properties
  - `remoteAddress` - remote IP address
  - `remotePort` - remote port number
  - `bufferSize` - size of `Buffer` that holds data to be written before it is sent
  - `bytesRead` - number of bytes read
  - `bytesWritten` - number of bytes written



# ... net.Socket Class ...

- Methods

- `connect(port, [host], callback)` - for TCP  
`connect(path, callback)` - for Unix socket
  - usually `net.connect` is used instead of this
  - might use this to implement a custom socket (by writing a new class that inherits `net.Socket`) or to reuse a closed `Socket` to connect to a different server
  - asynchronously opens a new connection
  - `host` defaults to localhost
  - returns nothing and passes nothing to callback
- `setEncoding(encoding)` - options are 'ascii', 'base64' and 'utf8' (default)
- `write(data, [encoding], [callback])`
  - encoding defaults to 'utf8', callback is invoked after all data has been written
- `end(data, [encoding])`
  - optionally writes more data; closes socket; server will receive 'end' event

# ... net.Socket Class ...

- Methods

- **pause()** - pauses reading of data; for throttling an upload
- **resume()** - resumes reading of data after a call to **pause()**
- **setTimeout(ms, [callback])**
  - invokes callback once if no reads or writes within ms
  - set to zero (default) for no timeout to wait forever and never invoke a callback
- **address()**
  - returns IP address and port of socket in a object with **address** and **port** properties
- **destroy()** - advanced
- **setNoDelay(bool)** - advanced
- **setKeepAlive(enable, [initialDelay])** - advanced

# ... net.Socket Class

- Events

- **connect** - when connection is established
- **data** - when data is received
  - callback is passed a **Buffer** or string containing the data
- **end** - when **end()** has been called on socket on other end
- **timeout** - when timeout occurs (see **setTimeout** method)
- **drain** - when write **Buffer** becomes empty
- **error** - when any socket-related error occurs
  - callback is passed an **Error** object
- **close** - when fully closed
  - callback is passed boolean indicating whether it was closed due to an error

# net Example

```
var net = require('net');  
var PORT = 8019;  
  
var server = net.createServer(function (socket) {  
    console.log('client connected'); 2  
  
    socket.on('data', function (data) {  
        console.log('received "' + data + '"'); 5  
    });  
  
    socket.on('end', function () {  
        console.log('client disconnected'); 7  
        server.close();  
    });  
  
    socket.write('hello');  
});  
  
server.on('error', function (err) {  
    console.error(err.code === 'EADDRINUSE' ?  
        'port ' + PORT + ' is already in use' :  
        err);  
});  
  
server.listen(PORT, function () {  
    console.log('listening on ' + PORT); 1  
});
```

Server

```
var net = require('net');  
  
var socket = net.connect(8019, function () {  
    console.log('connected to server'); 3  
});  
  
socket.on('data', function (data) {  
    console.log('received "' + data + '"'); 4  
    socket.write('goodbye');  
    socket.end();  
});  
  
socket.on('end', function (data) {  
    console.log('disconnected from server'); 6  
});
```

Client

	<b>Output from server</b>	<b>Output from client</b>
1)	listening on 8019	3) connected to server
2)	client connected	4) received "hello"
5)	received "goodbye"	6) disconnected from server
7)	client disconnected	

# Datagram

```
var dgram = require('dgram');
```

- User Datagram Protocol (UDP)
  - supports datagram sockets
- Datagram overview
  - messages are broken into packets
  - packets are separately addressed and routed
  - faster because it foregoes the handshaking overhead of TCP
  - doesn't guarantee reliability, packet ordering or data integrity
  - suitable when error checking and correction isn't needed or is provided by the application
  - suitable when dropping packets is better than waiting for them
- To create a datagram socket
  - `var dgs = createSocket(type, [callback])`
    - creates a datagram socket of a given type ('`udp4`' or '`udp6`')
    - optional callback gets '`message`' events (more in two slides)

**"I have a UDP joke to tell you, but you might not get it" ... unknown**

**Packet size** varies based on the Maximum Transmission Unit (MTU) of the transmission technology used.

For **IPv4** the minimum size is 68 bytes and the recommended size is 576 bytes.

For **IPv6** the minimum size is 1280 bytes.

Typically the actual packet size is at least 1500 bytes.

# Datagram Socket Methods

- **`dgs.send(buffer, offset, length, port, address, [callback])`**
  - sends a message that is in a specified chunk of a `Buffer` object
  - callback is passed `err` and number of bytes sent
- **`dgs.bind(port, [address])`**
  - starts listening on a given `port`
  - if `address` is specified, only listens on specified network interface instead of all
- **`dgs.close()`**
  - closes the datagram socket
- **`dgs.address()`**
  - gets address of socket in an object with `address` and `port` properties
- and more

see output from  
`os.networkInterfaces()`  
later

# Datagram Events

- **'message'**
  - when a message is received
  - callback is passed a `Buffer` and rinfo object with `address` and `port` properties
- **'listening'**
  - when socket begins listening
- **'close'**
  - when call to `close` method completes
- **'error'**
  - when an error occurs
  - callback is passed an `Error` object

# Datagram Server Example

```
var dgram = require('dgram');

var type = 'udp4'; // or 'udp6'
var server = dgram.createSocket(type);

server.on('message', function (msg, rinfo) {
  console.log('got "' + msg + '" from ' +
    rinfo.address + ':' + rinfo.port);

  msg = new Buffer('pong');
  server.send(msg, 0, msg.length, rinfo.port, rinfo.address, function (err, bytes) {
    console.log('bytes sent: ', bytes);
    server.close();
  });
});

server.on('error', function (err) {
  console.error(err);
});

server.on('listening', function () {
  var addr = server.address();
  console.log('listening on ' + addr.address + ':' + addr.port);
});

var PORT = 1234;
server.bind(PORT);
```

# Datagram Client Example

```
var dgram = require('dgram');

var type = 'udp4'; // or 'udp6'
var client = dgram.createSocket('udp4');

client.on('message', function (msg, rinfo) {
  console.log('got "' + msg + '" from ' +
    rinfo.address + ':' + rinfo.port);
  client.close(); // only expecting one message
});

client.on('error', function (err) {
  console.error(err);
});

client.on('listening', function () {
  var addr = client.address();
  console.log('listening on ' + addr.address + ':' + addr.port);
});

var msg = new Buffer('ping');
var HOST = 'localhost';
var PORT = 1234;
client.send(msg, 0, msg.length, PORT, HOST, function (err, bytes) {
  console.log('bytes sent: ', bytes);
});
```

Server Output

```
listening on 0.0.0.0:1234
got "ping" from 127.0.0.1:49617
bytes sent: 4
```

Client Output

```
listening on 0.0.0.0:49617
bytes sent: 4
got "pong" from 127.0.0.1:1234
```

automatically selected port

# Domain Name System (DNS)

```
var dns = require('dns');
```

- Resolves IP address from a domain name
  - `lookup` function
- Resolves domain name from an IP address
  - `reverse` function
- Retrieves many types of DNS records from a domain name
  - supported DNS record types are A (IPv4), AAAA (IPv6), CNAME (canonical name), MX (mail exchange), NS (name server), PTR (reverse IP lookup), TXT (text), SRV (service locator)
  - `resolve` function takes an array of DNS record types to retrieve
  - these functions return a specific type of DNS record:  
`resolve4, resolve6, resolveCname, resolveMx, resolveNs, resolveTxt, resolveSrv`
- For information on DNS record types,  
see [http://en.wikipedia.org/wiki/List\\_of\\_DNS\\_record\\_types](http://en.wikipedia.org/wiki/List_of_DNS_record_types)

# DNS Example

```
var dns = require('dns');

var domain = 'www.google.com';

dns.lookup(domain, function (err, address, family) {
  if (err) {
    throw err;
  }
  console.log(domain, address, 'IPv' + family);

  dns.reverse(address, function (err, domains) {
    if (err) {
      console.error('reverse lookup failed');
    } else {
      console.log(domains);
    }
  });
});
```

## Output

```
www.google.com 74.125.65.106 IPv4
[ 'gx-in-f106.1e100.net' ]
```

# HTTP

```
var http = require('http');
```

- Low-level API
- Typically the **express** module is used which builds on the **connect** module which builds on this
  - so we'll just cover the basics
- Supports streaming of requests and responses
  - rather than buffering until all the data is ready
- Use **querystring** core module to parse query parameters
  - covered in more detail later
- Can send HTTP requests with **http.request** function
  - userland module **request** is often used instead



# HTTP Example ...

```
var http = require('http');
var qs = require('querystring');

var PORT = 3002;

// Create an HTTP server and give it a 'request' listener.
var srv = http.createServer(function (req, res) {
  var url = req.url;

  // Many browsers, including Chrome, ask for this first.
  if (url === '/favicon.ico') {
    res.statusCode = 404;
    res.end(); // could also return an icon file and 200 status
    return;
  }

  console.log('method =', req.method);
  console.log('url =', url);
  console.log('headers =', req.headers);
  console.log('HTTP version =', req.httpVersion);

  var index = url.indexOf('?');
  var path = url.substring(0, index);
  console.log('path =', path);
  var queryString = url.substring(index + 1);
  var params = qs.parse(queryString); // can't pass entire URL
  console.log('query parameters =', params);
```

Sample output is based on browsing  
http://localhost:3002/foo/bar?  
month=April&color=yellow

see output  
two slides  
ahead

# ... HTTP Example ...

```
// Decide what to write in response based on path and query parameters.  
// Express supports defining "routes" which makes this easier.  
  
// If there is data in the request body, it can be received in chunks.  
var data = '';  
req.on('data', function (chunk) {  
    data += chunk;  
});  
req.on('end', function () {  
    // All the data has been received now.  
    console.log('data =', data);  
});  
  
var status = 200;  
var responseHeaders = {  
    'Content-Type': 'text/plain'  
};  
// Can set response status and other headers in one call.  
//res.writeHead(status, responseHeaders);  
  
// Can set response status and each header separately.  
res.statusCode = status;  
res.setHeader('Content-Type', 'text/plain');
```

chunk size is limited  
by TCP packet size

# ... HTTP Example

```
// Write the response body after all headers have been written.  
  
// Can write response body in one call.  
//res.end('Hello, World!');  
  
// Can write response body in chunks.  
res.write('Hello');  
res.write(', ');  
res.write('Chunks!');  
res.end();  
});  
  
srv.listen(PORT, function () {  
  console.log('ready');  
});
```

## Output

```
ready  
connection created  
method = GET  
url = /foo/bar?month=April&color=yellow  
headers = { host: 'localhost:3002',  
  'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:9.0.1) Gecko/20100101 Firefox/9.0.1',  
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',  
  'accept-language': 'en-us,en;q=0.5',  
  'accept-encoding': 'gzip, deflate',  
  'accept-charset': 'ISO-8859-1,utf-8;q=0.7,*;q=0.7',  
  connection: 'keep-alive',  
  'cache-control': 'max-age=0' }  
HTTP version = 1.1  
path = /foo/bar  
query parameters = { month: 'April', color: 'yellow' }  
data = nothing in request body
```

# Uniform Resource Locator (URL) ...

```
var url = require('url');
```

- Contains methods to resolve and parse URL strings
- **URL** object properties

- **protocol** - ex. 'http:'
- **auth** - ex. 'username:password'
- **host** - includes port; ex. 'company.com:3000'
- **port** - port number; ex. '3000'
- **hostname** - ex. 'company.com'
- **query** - query object; ex. {month: 'April', color: 'yellow'}
- **search** - query prepended with ?; ex. '?month=April&color=yellow'
- **pathname** - portion after **host** and before **search**; starts with a slash; ex. '/foo/bar'
- **path** - **pathname** plus **search**; ex. '/foo/bar?month=April&color=yellow'
- **hash** - fragment identifier; ex. '#baz'
- **href** - entire URL string;  
ex. 'http://username:password@company.com:3000/foo/bar?month=April&color=yellow#baz'

All examples below assume the full URL is 'http://  
username:password@company.com:3000/foo/bar?  
month=April&color=yellow#baz'

# ... URL

- Functions

- **parse(urlString, parseQueryString=false, slashesDenoteHost=false)**
  - creates and returns a `URL` object from a URL string
  - if `parseQueryString` is true, `query` property will be an object where keys are query parameter names and values are query parameter values;  
ex. { month: 'April', color: 'yellow' }
    - otherwise `query` property value is just the query string portion as a string without leading ?
  - if `slashesDenoteHost` is true, host will be obtained from first slashed part after //;  
`ex. url.parse('http://foo/bar/baz', false, true)` returns a URL object where `host = 'foo'`, `path = '/bar/baz'`, and `pathname` is the same
- **format(urlObject)**
  - takes a URL object and returns a URL string
- **resolve(from, to)**
  - returns a URL string created by treating `from` as the base URL and `to` as a relative URL
  - see example on next slide

# URL Example

```
'use strict';
var url = require('url');

var urlString =
  'http://username:password@company.com:3000/' +
  'foo/bar?month=April&color=yellow#baz';
var urlObj = url.parse(urlString, true, true);
console.log('urlObj =', urlObj);

urlObj.auth = 'fred:wilma';
urlObj.query.month = 'September';
urlObj.query.color = 'blue';
urlObj.hash = '#barney';
urlString = url.format(urlObj);
console.log('urlString =', urlString);

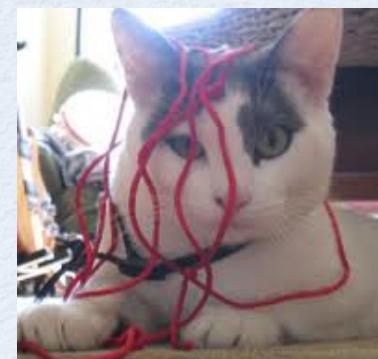
urlObj = { protocol: 'http:',
  slashes: true,
  auth: 'username:password',
  host: 'company.com:3000',
  port: '3000',
  hostname: 'company.com',
  href: 'http://username:password@company.com:3000/foo/bar?month=April&color=yellow#baz',
  hash: '#baz',
  search: '?month=April&color=yellow',
  query: { month: 'April', color: 'yellow' },
  pathname: '/foo/bar',
  path: '/foo/bar?month=April&color=yellow' }
urlString = http://fred:wilma@company.com:3000/foo/bar?month=April&color=yellow#barney
resolvedUrl = http://www.ociweb.com/knowledge-sharing/tech-com/sett'
```

Output

# Query Strings

```
var qs = require('querystring');
```

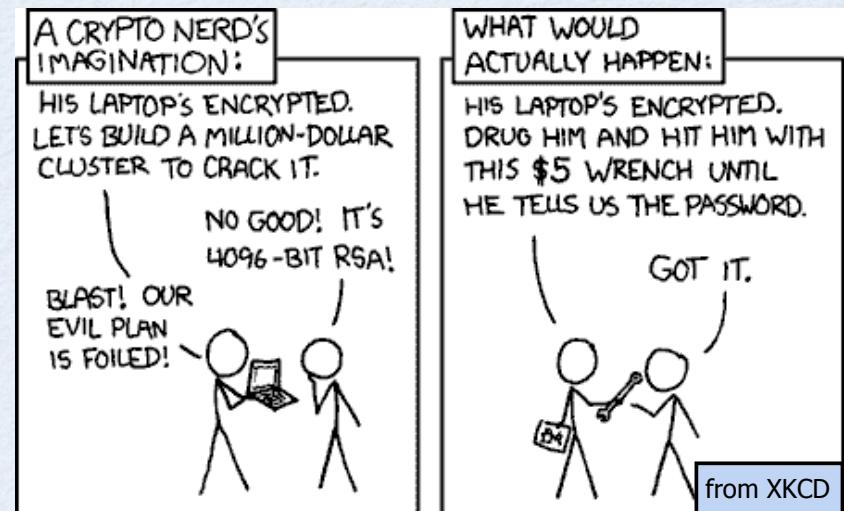
- Contains methods to parse and create query strings
  - part of URLs from ? to end
- Used by "url" module
- Functions
  - **stringify(obj, sep='&', eq='=')**
    - creates a query string from key/value pairs in obj
    - why would different delimiter characters ever be desired?
    - ex. `qs.stringify({month: 'April', color: 'yellow'})`  
returns `'month=April&color=yellow'`
  - **parse(str, sep='&', eq='=')**
    - creates an object containing key/value pairs from a query string
    - ex. `qs.parse('month=April&color=yellow')`  
returns `{ month: 'April', color: 'yellow' }`
  - other functions are mainly for internal use



# Crypto

```
var crypto = require('crypto');
```

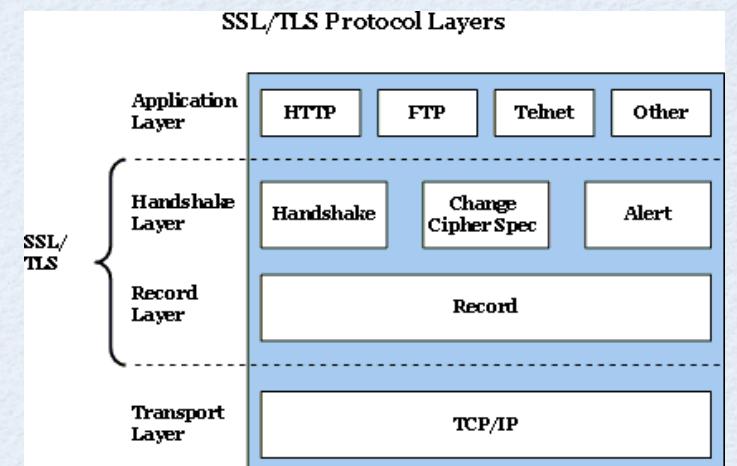
- Provides functions for working with security credentials that are used with HTTP and HTTPS
- Works with concepts such as
  - Privacy Enhanced Email (PEM) credential
  - cryptographic hash
  - digest
  - Hash-based Message Authentication Code (HMAC)
  - cipher / decipher
  - signer object
  - verification object
  - Diffie-Hellman key exchange
  - asynchronous PBKDF2
- Relies on OS `openssl` command
- Beyond my knowledge to say anything further



# TLS/SSL

```
var tls = require('tls');
```

- Secure Socket Layer (SSL)
- Transport Layer Security (TLS) - an upgrade to SSL 3.0
- Both are cryptographic protocols for secure internet communication
  - public/private key infrastructure
  - prevents eavesdropping and tampering with message content
- Functions
  - `tls.createServer(options, [connectionListener])`
    - called by server code
    - options include key, certificate and certificate authority (CA) file contents
      - can also set `rejectUnauthorized` option to `true` to reject connections not authorized by a CA in list of authorized CAs
    - returns a `tls.Server` object (see next slide)
  - `tls.connect(port, [host], [options], [connectionListener])`
    - called by client code
    - options include key, certificate and certificate authority (CA) file contents
    - returns a `tls.CleartextStream` object (see next slide)



# TLS Classes

- **tls.Server**
  - type of object returned by `tls.createServer()`
  - “a subclass of `net.Server` and has the same methods”
    - including `listen(port)`
  - “Instead of accepting just raw TCP connections, this accepts encrypted connections using TLS or SSL.”
- **tls.CleartextStream**
  - type of object returned by `tls.connect()`
  - has same methods and events as readable and writable streams
  - “a stream on top of the encrypted stream that makes it possible to read/write an encrypted data as a cleartext data”

# TLS Example Server

```
var fs = require('fs');
var tls = require('tls');

var opts = {
  key: fs.readFileSync('mykey.pem'),
  cert: fs.readFileSync('mycert.pem'),
};

var server = tls.createServer(opts, function (cts) {
  console.log('server connected',
    cts.authorized ? 'authorized' : 'unauthorized');

  cts.setEncoding('utf8');           if encoding isn't specified then data
                                    will be a Buffer instead of a string
                                    (no default encoding)
  cts.write('ping');
  cts.on('data', function (data) {
    console.log('got', data, 'from client');
    server.close();
    process.exit(0);
  });
  cts.on('end', function () {
    console.log('got end event from client');
  });
});

server.listen(8000, function() {
  console.log('ready');
});
```

see HTTPS section (slide 51) for command  
to generate key and certificate .pem files

will only get 'authorized'  
if `rejectUnauthorized`  
option is `true`

# TLS Example Client

```
var fs = require('fs');
var tls = require('tls');

var opts = {
  key: fs.readFileSync('mykey.pem'),
  cert: fs.readFileSync('mycert.pem'),
};

var cts = tls.connect(8000, opts);
cts.setEncoding('utf8');
cts.on('secureConnect', function () {
  console.log('client connected',
    cts.authorized ? 'authorized' : 'unauthorized');
});
cts.on('data', function (data) {
  console.log('got', data, 'from server');
  cts.write('pong');
});
cts.on('end', function () {
  console.log('got end event from server; process will exit');
});
cts.on('error', function (e) {
  var msg = e.code === 'ECONNREFUSED' ?
    'failed to connect; is server running?' : e.toString();
  console.error(msg);
});
```

can also pass this callback to `tls.connect()`; this will be set to the `cts` object inside it

if encoding isn't specified then `data` will be a `Buffer` instead of a string (no default encoding)

# TLS/SSL Advanced Functionality

- Start a TLS session on an existing TCP connection
- Next Protocol Negotiation (NPN)
  - TLS handshake extension to use one TLS server for multiple protocols (HTTP and SPDY)
- Server Name Indication (SNI)
  - TLS handshake extensions to use one TLS server for multiple hostnames with different SSL certificates

# HTTPS

```
var https = require('https');
```

- HTTP over SSL/TLS
  - Secure Socket Layer (SSL) preceded Transport Layer Security (TLS)
  - these are cryptographic protocols
  - from Wikipedia, “encrypt the segments of network connections above the Transport Layer, using asymmetric cryptography for key exchange, symmetric encryption for privacy, and message authentication codes for message integrity”
- Need .pem files for key and certificate
  - “Privacy Enhanced Mail”
  - one way to create is to run following command and answer prompts

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout mykey.pem -out mycert.pem
```

type of certificate	makes key	valid for this
signing request	unencrypted	many days
no password required to use		
- **https.Server** is a subclass of **tls.Server**
- **https.request** function sends a request to a secure web server
  - supports same options as **http.request** function
  - userland module **request** is often used instead



# HTTPS Example

```
var fs = require('fs');
var https = require('https');

var PORT = 3002;
var opts = {
  key: fs.readFileSync('mykey.pem'),
  cert: fs.readFileSync('mycert.pem')
};

var srv = https.createServer(opts, function (req, res) {
  // Many browsers, including Chrome, ask for this first.
  if (req.url === '/favicon.ico') {
    res.statusCode = 404;
    res.end(); // could also return an icon file and 200 status
    return;
  }

  res.statusCode = 200;
  res.end('Hello, World!');
});

srv.listen(PORT, function () {
  console.log('ready');
});
```

browse with `https://localhost:3002` or  
run "`curl -k https://localhost:3002`"  
(-k allows SSL connections without certificates)

# Virtual Machine (VM) ...

```
var vm = require('vm');
```

- Compiles a string of JavaScript code and runs it or saves it so it can be run later without recompiling
- The code does not have access to variables in local scope, regardless of the `vm` function used
  - to run code that can access variables in local scope, use the JavaScript `eval` function
- Syntax errors in the code string passed to these functions are reported to stderr and an `Error` is thrown
- Functions in this module that run code return the value of the last expression evaluated
  - `return` statements cannot be used in the top-level of a code string, only inside function definitions within a code string

# ... VM

## • Functions

- `runInThisContext(code, [filename])`
  - global object for code is current global object; assign properties to `global` to make them accessible
- `runInNewContext(code, [sandbox], [filename])`
  - global object for code is sandbox object
  - creates a new context which has overhead
- `createContext([sandbox])`
  - creates a `Context` object that can be passed to `vm.runInContext()`
- `runInContext(code, context, [filename])`
  - context must be created by calling `vm.createContext(sandbox)` (see next slide)
  - global object for code is sandbox object passed to `createContext`
  - context object also holds built-in objects and functions
  - **more efficient than `runInNewContext` when the same context will be used multiple times**
- `createScript(code, [filename])`
  - compiles code and returns a `Script` object that can be used execute the code later
  - see `Script` methods on next slide

The optional filename arguments appear in stack traces to help with debugging.

# VM Script Class

- Objects created by calling `vm.createScript(code)`
- Methods
  - `runInThisContext()`
    - global object for code is current global object
    - assign properties to `global` to make them accessible
  - `runInNewContext([sandbox])`
    - global object for code is sandbox object

# VM Example

```
var assert = require('assert');
var vm = require('vm');

var code = "Math.pow(x, y)";
var code2 = "console.log('z =', z); " + code;

console.log('global =', global); // has lots of variables and functions
global.x = 3;
global.y = 2;
global.z = 19;
// Note how global functions (in this case just console)
// are explicitly being made available in the sandbox and context.
var sandbox = {x: 2, y: 3, z: 19, console: console};
var context = vm.createContext({x: 2, y: 4, z: 19, console: console});

assert.equal(vm.runInThisContext(code), 9); // 3 ^ 2
assert.equal(vm.runInNewContext(code2, sandbox), 8); // 2 ^ 3
assert.equal(vm.runInContext(code2, context), 16); // 2 ^ 4

var script = vm.createScript(code);
assert.equal(script.runInThisContext(), 9); // 3 ^ 2
assert.equal(script.runInNewContext(sandbox), 8); // 2 ^ 3
assert.equal(script.runInContext(context), 16); // 2 ^ 4
```

# Child Processes

```
var cp = require('child_process');
```

- **Functions** [all of these return a `ChildProcess` object]

- **`spawn(command, args=[], [options])`**
  - starts a new process that runs a given command and returns a `ChildProcess` object
  - `args` holds command-line flags and arguments
  - `cwd` option specifies directory in which command runs (defaults to current)
  - `env` option specifies environment variables available in child process (defaults to `process.env`)
  - to obtain output, listen for `data` events on `stdout` and `stderr` properties
- **`exec(command, args=[], options, callback)`**
  - runs a command in a shell, buffers output to `stdout` and `stderr`, and passes it to a callback function of the form `function (err, stdout, stderr)`
  - supports a `timeout` option
  - callback is passed status code, `stdout Buffer` and `stderr Buffer`
- **`execFile(file-path, args=[], [options], callback)`**
  - executes commands in specified file in current process
  - callback is passed status code, `stdout Buffer` and `stderr Buffer`
- **`fork(script-path, args=[], options)`**
  - similar to `spawn`, but returned object has a `send` method that emits '`message`' events

process doesn't end when end of script is reached; must call `process.exit()` in script

# ChildProcess Class

- Inherits from **EventEmitter**
- Events
  - **exit** - emitted after child process ends
    - callback function takes a status code and a signal
    - a code is passed on normal termination
    - a signal is passed if terminated by a signal
- Properties
  - **stdin** - standard input stream
  - **stdout** - standard output stream
  - **stderr** - standard error stream
  - **pid** - process id
- Methods
  - **send(*message*)**
    - sends message to child process
  - **kill(*signal='SIGTERM'*)**
    - sends a given signal to the child process

# Child Process Example #1

```
var child_process = require('child_process');

var cp = child_process.spawn(
  'ls', ['-l', '..']);
console.log('pid =', cp.pid); runs the "ls -l" command
                                in the parent directory

cp.stdout.on('data', function (data) {
  console.log('data =', data.toString());
});

cp.on('exit', function (code, signal) {
  console.log('exit code =', code);
  console.log('exit signal =', signal);
});
```

Output

```
pid = 16511
total 0
drwxr-xr-x  7 Mark  staff  238 Jan 28 18:36 addons
drwxr-xr-x  4 Mark  staff  136 Dec  7 20:52 async
drwxr-xr-x  3 Mark  staff  102 Nov 21 08:50 buffers
drwxr-xr-x  5 Mark  staff  170 Nov 15 15:03 callbacks
drwxr-xr-x  8 Mark  staff  272 Feb 18 14:04 child_process
...
drwxr-xr-x  5 Mark  staff  170 Jan   8 13:19 vm
drwxr-xr-x  4 Mark  staff  136 Feb 15 18:38 zlib

exit code = 0
exit signal = null
```

# Child Process Example #2

finds every required module in the .js files  
in a below the parent directory

```
var child_process =
  require('child_process');

var args = ['js', 'require'];
var opts = {cwd: '..'};
var file = 'child_process/myFind.sh';
var cp = child_process.execFile(file, args, opts, function (err, data) {
  if (err) {
    return console.error(err);
  }

  var re = /require\(['"](.*)['"]\)\//;
  var requires = {};
  data.split('\n').forEach(function (line) {
    var matches = re.exec(line);
    if (matches) {
      requires[matches[1]] = true;
    }
  });
  Object.keys(requires).sort().forEach(function (req) {
    console.log(req);
  });
});
```

```
#!/bin/bash
# Finds all files with a given file extension
# in and below the current directory
# that contain a given string.
# For example, myFind java "implements Foo"

if [ $# -ne 2 ]; then
  echo usage: myFind {file-extension} {search-string}
  exit 1
fi

find . -name "*.$1" | xargs grep "$2"
```

Output

```
./lib/math
./build/Release/demo
./build/Release/hello
./demo1
./helper
assert
async
child_process
...
util
vm
zlib
```

# Cluster

```
var cluster = require('cluster');
```

- “easily create a network of processes that all share server ports”
  - works with any TCP-based server, including HTTP and HTTPS
- Builds on “Child Processes” module
- Initial process is called “master”
  - only process that listens on selected port
  - uses inter-process communication (IPC) pipes to communicate with workers
- Forked processes are called “workers”
  - typically want to fork a number of workers not greater than number of processors
  - get number of processors with `os.cpus().length`
  - no guarantees about order of selection of workers to handle requests
    - distributes connections across workers, but doesn’t distribute requests
    - once a client gets a connection, all their requests will go to the same worker

“The difference between `cluster.fork()` and `child_process.fork()` is simply that cluster allows TCP servers to be shared between workers.

`cluster.fork` is implemented on top of `child_process.fork`.

The message passing API that is available with `child_process.fork` is available with cluster as well.”

“The Jewel Box (also known as NGC 4755, the Kappa Crucis **Cluster** and Caldwell 94) is an open cluster in the constellation of Crux.” ... Wikipedia



# Cluster Masters

- Can fork workers
- Can send messages to workers
  - `worker.send('message');`
- Can listen for messages from workers
  - `worker.on('message', function (obj) {...});`
- Can listen for death of workers
  - `cluster.on('death', function (worker) {...});`
  - anything that kills the process triggers this, including OS `kill` command and throwing an uncaught exception
  - can optionally fork replacement workers
    - typically the only job of master after it forks workers

# Cluster Workers

- Have a unique id
  - in `process.env.NODE_WORKER_ID` within their process
- Typically start a server such as an HTTP server
- Can send messages to their master
  - `process.send(obj);`
- Can listen for messages from master
  - `process.on('message', function (msg) { ... });`
- Cannot send messages to other workers
- Cannot fork more workers
- Are killed if their master dies

# Cluster Example ...

```
var cluster = require('cluster');

if (cluster.isMaster) { // cluster.isWorker is also set
    var requestCount = 0;
    var handleMsg = function (msg) {
        if (msg.cmd === 'gotRequest') {
            requestCount++;
            console.log('requestCount =', requestCount);
        }
    };

    cluster.on('death', function (worker) {
        console.log('worker with pid', worker.pid, 'died - starting new worker');
        worker = cluster.fork();
        worker.on('message', handleMsg);
    });

    // Fork worker processes.
    var cpuCount = require('os').cpus().length;
    for (var i = 1; i < cpuCount; i++) {
        var worker = cluster.fork();
        worker.on('message', handleMsg);
    }
}
```

same code is run  
for the master  
and each worker

# ... Cluster Example

```
} else { // for workers
  // Start an HTTP server in worker processes.
  var http = require('http');
  var PORT = 8000;
  http.Server(function (req, res) { // not a constructor function
    if (req.url === '/favicon.ico') {
      res.writeHead(404);
      res.end(); // could also return an icon file and 200 status
      return;
    }

    // Simulate taking a while to process request.
    setTimeout(function () {
      res.statusCode = 200;
      res.end('Hello from process ' + process.pid + '!\n');

      console.log('worker with pid', process.pid, 'handled a request');

      // Send message to master process.
      process.send({cmd: 'gotRequest'});
    }, 1000); // one second
  }).listen(PORT);

  var workerId = process.env.NODE_WORKER_ID; // numbered starting from 1
  console.log('worker server', workerId, 'ready, pid', process.pid);
}
```

1. browse `http://localhost:8000`
2. kill the process that handled the request
3. refresh the page and note that a different process handles the request

# Recommended Userland Modules ...

- **Async**
  - provides many functions that simplify writing asynchronous code
    - operations on collections, including functional ones like map and reduce
    - control flow
  - can be used with Node and in browsers
- **DNode**
  - asynchronous, OO RPC system for calling remote functions
- **express**
  - HTTP request routing and redirection
  - view rendering
  - built on Connect
- **Formidable**
  - parses form data, especially file uploads
- **Jade**
  - HTML template engine, similar to HAML
- **mkdirp**
  - Unix “`mkdir -p`” for Node
- **Moment.js**
  - parses, manipulates and formats dates
- **mongo-native**
  - Node driver for MongoDB
- **Optimist**
  - command-line option parsing

# ... Recommended Userland Modules

- Request
  - constructs and sends HTTP requests
- Rimraf
  - Unix “`rm -rf`” for node
- Socket.IO
  - realtime messaging between Node and all browsers
- sprintf
  - C “`sprintf`” function for node
- Stylus
  - translates an alternative CSS syntax to standard CSS
  - similar to Sass and LESS
- Threads A GoGo
  - creates and runs JavaScript threads for asynchronous processing within a single Node process
- Underscore
  - large collection of generally useful utility functions

# Node.js Resources

- Main site - <http://nodejs.org/>
- API doc - <http://nodejs.org/docs/latest/api/>
- NPM Registry Search - <http://search.npmjs.org/>
- How To Node - <http://howtonode.org/>
- node-toolbox - <http://toolbox.no.de/>
- NodeUp podcast - <http://nodeup.com/>
- Felix Geisendoerfer's guide - <http://nodeguide.com>