modules in Node

We've already seen the usage of some Node modules:

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

Creating a custom module:

```
var message = function (message) {
    console.log(message);
};

module.exports = message;
```

To use our custom module:

```
var messenger = require('./custom_messenger');

messenger('Hello, class!');
```

Another way to export our custom module:

```
var messenger = function (message) {

    console.log('message');
};

module.exports.messenger = messenger;
```

To use this module:

```
var msg = require('./custom_messenger_alt');

msg.messenger('Hello, class!');
```

or, alternatively:

```
require('./custom_messenger_alt').messenger('Hello, class!');
```

exporting multiple methods:

```
var message_log = function (message) {

    console.log('LOG: ' + message);
};

var message_warn = function (message) {

    console.log('WARN: ' + message);
};

var message_error = function (message) {

    console.log('ERROR: ' + message);
};

var message_preamble = function () {

    console.log('here is a preamble');
};

module.exports.log = message_log;
module.exports.warn = message_warn;
module.exports.error = message_error;
```

exporting an object with multiple methods - this is similar to the classical module pattern:

```
var message_log = function (message) {
    console.log('LOG: ' + message);
};

var message_warn = function (message) {
    console.log('WARN: ' + message);
};

var message_error = function (message) {
    console.log('ERROR: ' + message);
};

var message_preamble = function () {
    console.log('here is a preamble');
};

var logger = {
    log: message_log,
    warn: message_warn,
    error: message_error
};

module.exports = logger;
```

using them:

```
var msg = require('./messenger_xl');

msg.log('here is just a logging message');
msg.warn('thar be dragons!');
msg.error('critical failure');
```

how Node resolves required modules

how Node resolves required modules

```
// look in the same directory
var messenger = require('./messenger.js');

// look in the parent directory
var messenger = require('../messenger.js');

// look in the path specified
var messenger = require('/usr/local/modules/custom');
```

modules without path specified:

```
// start from the current directory, looking inside node_modules
var messenger = require('messenger');

// then keep going up one parent directory, looking for node_modules

// until finally checking the root directory for node_modules
```

inside the node_modules directory will be more directories, each module installing their own

npm

npm is the package manager for Node modules that comes with Node

npmjs.com is also the repository for Node modules that can be installed via npm - as of this date, there are over 260k modules publicly available

npm has dependency management built in

to install a module for a local project:

```
npm install request
```

to install modules globally:

note: you can't require globally installed modules in code - they must still be installed locally

```
npm install -g grunt
```

when building a Node project, you will also want to create a package.json file for your project, located inside your project directory

app/package.json

```
{
    "name": "Goats for Granny",
    "version": "1",
    "dependencies:" {
        "underscore" "1.8.3",
        "handlebars": "3.0.1"
    }
}
```

package.json allows you to install all dependencies

```
// from the project directory where package.json is found
npm install
```

dependencies will be installed into a node_modules directory in your project folder. It used to recursively find package.json files in those directories to install that module's dependencies, but as of npm 3, it now uses a flat folder system for installing downstream module dependencies

exports vs module.exports

- your module returns module.exports, not exports
- exports collects properties and attaches them to module.exports if module.exports is not already defined
- if module.exports has been defined, exports is ignored

semantic versioning

semantic versioning example:

Major: 1

Minor: 8

Patch: 3

```
"underscore": "1.8.3"
```

Semantic Versioning 2.0.0

1. MAJOR version when you make incompatible API changes (will break things)
2. MINOR version when you add functionality in a backwards-compatible manner (probably won't break things)
3. PATCH version when you make backwards-compatible bug fixes (shouldn't break anything)

versioning ranges:

```
/*
 * dangerous - allows anything greater than or equal to 1.0.0,
 * but less than 2.0.0 - minor version difference can be breaking
 * /
"underscore": "~1"

/*
 * not as dangerous - allows anything greater than or equal to 1.8.0,
 * but less than 1.9.0 - this could still represent a substantial change
"underscore": "~1.7"

/*
 * safe - allows anything greater than or equal to 1.8.3, but less
 * than 1.9.0 - only a few patch changes possible
 * /
"underscore": "~1.8.3"
```