the NODE FIRM

STREAMS

STREAMS PRODUCE AND CONSUME DATA

Data can be a buffer, a string, or an object.
Buffers are the default data type for streams.

Streams can wrap resources like a file, a TCP connection or event the body on an HTTP message.

Every object that can emit data or receive data is a candidate for being a Stream.

```
res.on('data', function(d) {
  console.log('chunk with %d bytes', d.length);
});
```

THE BASE STREAMS:

- Readable Stream
- Writable Stream

READABLE STREAM

Base class for objects that you can read data from.

Examples:

- reading a file
- reading data from a TCP server
- the body data on an incoming HTTP request
- many others

READABLE EVENT

Listen to `readable` to know when data is available for comsumption.

```
stream.on('readable', function() {
  var data;

  while (data = stream.read()) {
    console.log('I have some data:', data);
  }
});
```

SET THE ENCODING

By default, `stream.read()` returns a Buffer object.
To decode these buffers into strings, specify an encoding:

```
stream.setEncoding('utf8');
```

Supported encodings: `utf8`, `utf16le`, `ucs2`, `ascii`, and `hex`

THE END EVENT

When the stream ends it emits the `end` event:

```
stream.once('end', function() {
  console.log('stream ended!');
});
```

FLOW CONTROL

A readable stream buffers the data read from the underlying resource up until the high water mark is reached.

The default for `highWaterMark` is 16kb. This is tunable.

EXAMPLES

Reading a file

01_read_file.js:

```
var fs = require('fs');
var stream = fs.createReadStream(__dirname + '/support/npm.json');

stream.on('readable', function() {
  var data;
  while (data = stream.read()) {
    console.log('I have a piece with %d bytes', data.length);
  }
});
```

An HTTP client response

02_read_http_response.js:

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

http.get('http://www.google.com/search?q=node.js', function(res) {
  res.setEncoding('utf8');

  res.on('readable', function() {
    var data;
    while(data = res.read()) {
      console.log('[chunk] %s', data);
    }
  });

  res.once('end', function() {
    console.log('[ended]');
  });
});
```

WRITABLE STREAM

Base class for objects you can write data to.

Examples:

- a file write stream
- an http client request
- a TCP connection to a TCP server
- many others

STREAM.WRITE()

A writable stream offers a `write()` method.

```
stream.write(buffer);
```

The buffer can be a raw buffer, a string.

WRITING STRINGS

Writing a string, the default encoding is `utf8`.
You can specify a different encoding as the second parameter.

```
stream.write('9bc61ad8', 'hex');
```

FLOW CONTROL

Provide a callback to notified when data has been successfully written to the underlying resource.

```
stream.write('Hey!', function() {
  console.log('wrote it');
});
```

STREAM BACKPRESSURE

`stream.write` returns a boolean specifying whether the buffer was flushed.

- written to the network
- written to disk

BUFFERING

A `false` return value indicates that the last write is buffered in memory.
Continuing to send more data will bloat your process' memory usage

DRAINED

Once the underlying resource has caught up, a `"drain"` event will be emitted.

EXAMPLE

Writing a file

03_write_file.js:

```
var fs = require('fs');
var times = parseInt(process.argv[2]) || 1;
var stream = fs.createWriteStream(__dirname + '/out.txt');

console.log('Times: %d', times);

stream.on('drain', function() {
  process.stdout.write('D');
});

var interval = setInterval(function() {

  for (var i = 0; i < times; i++) {
    var flushed = stream.write(new Buffer(1024));
    process.stdout.write(flushed ? 'F' : 'Q');
  }
  process.stdout.write('#');
}, 100);

setTimeout(function() {
  clearInterval(interval);
  stream.end();
}, 2000);
```

The passed argument determines how much data in kilobytes to write per write cycle

```
$ node 03_write_file.js 50
```

Increasing this number, you will eventually see a repeated pattern like:

```
FFFFQQQQQQQQQQQQQQQQ#D
```

- writes start out by being flushed ('F')
- buffers fill up and they start getting queued ('Q')
- cycle ends ('#')
- Later, the queue drains ('D')

PIPING

You can connect a readable stream to a writable stream using pipe:

```
source.pipe(target);
```

PIPE BUILT IN FLOW CONTROL

- Stops reading from the source if the target doesn't flush.
- Starts again once the target resumes.

Configuring the end behavior

```
source.pipe(target) // will end source when target ends
source.pipe(target, { end: false }); // doesn't end source when target e
```

EXAMPLE

Downloading the Node.js logo from the web into a file.

04_pipe.js:

```
var get = require('http').get;
var fs = require('fs');

var target = fs.createWriteStream(__dirname + '/support/nodejs.png');

get('http://nodejs.org/logo.png', function(res) {
  res.pipe(target);
});

target.once('finish', function() {
  console.log('done');
});
```

CHAINING

Pipes are chainable. `stream.pipe` returns the target stream.
This means you can chain streams together like this:

```
source.pipe(transform).pipe(target);
```

Example of a chatty server:

05_chaining.js:

```
var server = require('net').createServer();

server.on('connection', function(stream) {
  process.stdin.pipe(stream).pipe(process.stdout);
});

server.listen(8000, function() {
  console.log('listening on port 8000');
});
```

UNPIPING

If you want to stop a pipe at any time, you can use the `stream.unpipe` method:

```
source.unpipe(target);
```

Or you can cancel **all** the pipes on the source:

```
source.unpipe();
```

DUPLEX STREAM

A duplex stream is a stream that is both readable and writable, such as a TCP connection.

EXAMPLE

Echo server

06_duplex.js:

```
var server = require('net').createServer();

server.on('connection', function(stream) {
  stream.pipe(stream);
});

server.listen(8000, function() {
  console.log('listening on port 8000');
});
```

TRANSFORM STREAM

A transform stream is also a readable and writable stream, but act locally on the data, transforming it.

TRANSFORM STREAM REQUIREMENTS

- specify the `_transform(buf, encoding, done)` method
- push transformed data back into the stream using `this.push(data)`

UPPERCASE TRANSFORM EXAMPLE

A transforms a stream into upper case

07_transform.js:

```
var Transform = require('stream').Transform;

function createTransformStream() {
  var upcase = new Transform();
  upcase._transform = function _transform(buf, encoding, done) {
    this.push(buf.toString().toUpperCase());
    done();
  };
  return upcase;
}

process.stdin.pipe(createTransformStream()).pipe(process.stdout);
```

ONE MORE EXAMPLE

Here is an example where we:

- create an http request
- pipe the result to the zlib/Gunzip library
- pipe the next result to the tar/Extract library
- pipe the next result to the filestream
- if there is a file, console.log it
- when done, console.log

NPM install `tar` and `request`:

```
$ npm install tar request
```

08_example.js:

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

/// create streams

/// plumbing
```

08_example.js:

```
/// create streams

// create download stream
var download = request('https://github.com/dscape/p/tarball/master');

download.once('end', function() {
  console.log("download done");
});

// create gunzip transform stream
var gzip = zlib.Gunzip();

// create extract write stream
var extractor = tar.Extract({ path: __dirname + '/support' });

extractor.on('entry', function(entry) {
  if (entry.type === 'File') {
    console.log('[unpack] `' + entry.path + '`');
  }
});

extractor.on('end', function () {
  console.log("unpack done");
});
```

08_example.js:

```
/// plumbing

// download -> gzip -> extractor
download.pipe(gzip).pipe(extractor);

// download -> ./support/p.tar.gz
download.pipe(fs.createWriteStream(__dirname + '/support/p.tar.gz'));
```

SUMMARY

- Streams provide a mechanism to expressively manipulate large amounts of data
- Data potentially much larger than V8 could hold in memory
- Flexible and easy to implement
- Composable
- Large number of third-party modules with stream support