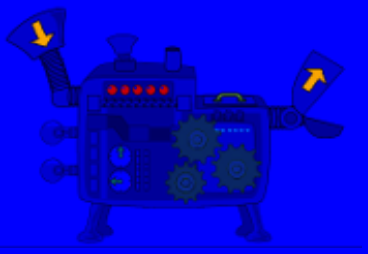


JavaScript input & output

Writing to stdout


```
"use strict";
console.log("Hello World!");
process.stdout.write("Hello ");
process.stdout.write("World!\n");
```

```
js> node {f}
Hello World!
Hello World!
js>
```

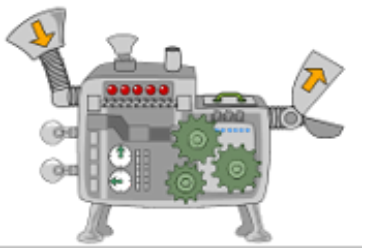
Stdout

The `console.log` facility is a formatting wrapper around the raw output stream `process.stdout`

On the Web, `console.log` exists but not `process.stdout`

An output stream only has methods `write` and `end`

You can write either text or binary data

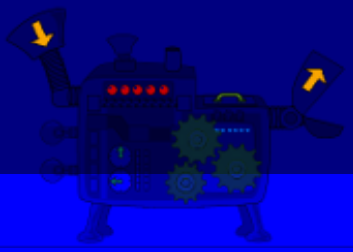


Reading from stdin


```
"use strict";
process.stdout.write("> ");
process.stdin.on('data', receive);

function receive(x) {
  console.log("You typed:", x);
  process.stdout.write("> ");
}
```

```
js> node {f}
> abc
You typed: <Buffer 61 62 63 0a>
> def
You typed: <Buffer 64 65 66 0a>
>
```



Stdin

Using `stdin` involves event handling

`process.stdin.on('data', receive)` sets up `receive` as an event handler for data arrival

The `receive` function is passed as an argument, ***without*** calling it, as a ***callback function***

The data arrives as binary

The program doesn't stop until `^C` (or `^D` or `^Z`) is typed, because events are still possible

Receiving as text


```
"use strict";
process.stdout.write("> ");
process.stdin.setEncoding('utf8');
process.stdin.on('data', receive);

function receive(x) {
  console.log("You typed:", x);
  process.stdout.write("> ");
}
```

```
js> node {f}
> abc
You typed: abc

>
```

setEncoding

The call `setEncoding` specifies that text is expected

It also specifies the encoding, with `utf8` being the favourite (ascii-compatible Unicode)

The text that arrives ***includes*** the newline

Island adventure


```
// The island game
"use strict";

// Set up the world and current place
var spade = { what:"a spade", take:take };
var sand = { what:"sand", dig:dig };
var tree = { where:"near a tree", go:go, spade:spad
var beach = { where:"on a beach", go:go, north:tree
tree.south = beach;
var here = tree;

// Test if the player is carrying a given thing
function isBeingCarried(thing) {
  return (thing.drop != undefined);
}

// Define the actions
function take() {
```

Playing island adventure


```
js> node p8.js
You are near a tree
There is a spade here
> go south
You are on a beach
There is sand here
> dig sand
You have no spade
> go north
You are near a tree
There is a spade here
> take spade
You are carrying a spade
> go south
You are on a beach
There is sand here
> dig sand
You find your holiday return ticket and go home
js>
```

What now?

That completes our island adventure game for now, unless we want to network it later

We are going to have a look at reading and writing files, and use that to investigate:

- modules
- function notation
- callback details (closures)
- concurrency

Writing a file


```
"use strict";
var fs = require("fs");

var text = "Line one\nLine two\n";
fs.writeFile("out.txt", text, finish);

function finish(err) {
  console.log(err);
}
```

```
js> node {f}
null
js>
```

Modules

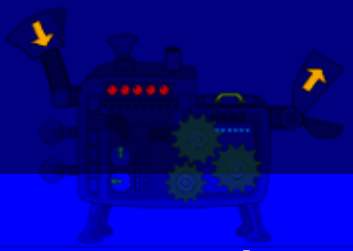
The call `require("name")` loads an installed module, with `fs` being the file system module

`require("./name")` loads a local module

The result is an object, with methods attached

The `node.js` module system is not like JavaScript on the Web (until version 6 arrives)

It is justified by putting it in a (non-standard) library, and by the fact that the scheme can (just about) be simulated on old JavaScript systems



Web modules

On the Web, everything defined in a module is global, and gets mixed in with, or clashes with, the globals from every other module, so it is normal to define a single function with everything in it (even `"use strict"`!)

```
Mod();
function Mod() {
  "use strict";
  Mod.fun = fun;
  function fun() { ... }
  ...
}
```

All variables and methods are attached to the single function

writeFile

The `fs.writeFile` function writes a whole file in one go, but you can of course also use open/write/close

It is ***asynchronous***, i.e. it returns immediately before the file has been completely written

You find out when the file has been written (or if something goes wrong) via a callback

An inline variant


```
"use strict";
var fs = require("fs");

var text = "Line one\nLine two\n";
fs.writeFile("out.txt", text, function(err) {
  console.log(err);
});
```

```
js> node {f}
null
js>
```

Function notation

The expression `function(args) { code }` is an anonymous function

In my personal opinion

1. Everybody uses this notation, it is ***essential*** to understand it
2. With very few exceptions, the notation is an ***abomination***, good programmers shouldn't use it

Why not?


```
f(...,... function() {
  ...
});
```

This has a block (vertical) inside an expression (horizontal), and the last line `});` is extremely ugly

In the end, the issue is readability and personal preference

I think the practice is common because of the ***stupid*** convention on the Web that everything is global, so people (rightly) try to avoid defining global variables

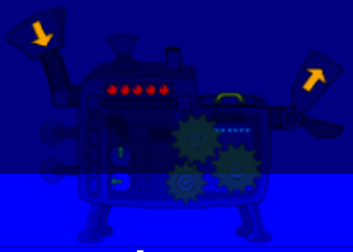
Reading a file


```
"use strict";
var fs = require("fs");

fs.readFile("in.txt", "utf8", finish);

function finish(err, text) {
  process.stdout.write(text);
}
```

```
js> node {f}
Line one
Line two
js>
```



readFile

The `readFile` function reads a whole file, and `utf8` says text, not binary

You can also use open/read/close, to read a file in chunks

It is asynchronous, returning straight away, and the text is passed to a callback

Next, let's have a go at writing a program that copies one file to another

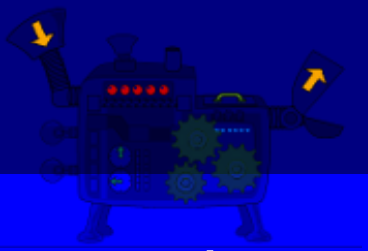
First attempt at copying


```
"use strict";
var fs = require("fs");

function copy(file1, file2) {
  fs.readFile(file1, "utf8", ready);
}

function ready(err, text) {
  console.log("Got text, what now?");
}
copy("in.txt", "out.txt");
```

```
js> node {f}
Got text, what now?
js>
```



The problem

In the `copy` function, after calling `readFile`, there is nothing that can be done because the text isn't ready

In the `ready` function, the next thing to do is to write to `file2`, but `file2` isn't available

Can we arrange for `file2` to be passed to `ready`? No

Should we put `file2` into a global variable so `ready` can pick it up? No, no, no!

The solution is coming up next:

Proper copy


```
"use strict";
var fs = require("fs");

function copy(file1, file2) {
  fs.readFile(file1, "utf8", ready);
  function ready(err, text) {
    fs.writeFile(file2, text, end);
  }
  function end(e) { console.log("done"); }
}
copy("in.txt", "out.txt");
```

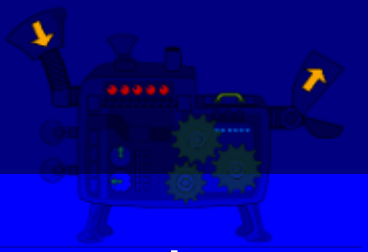
```
js> node {f}
done
js>
```

Inner functions

The secret to making callbacks work in JavaScript is to define one function `g` inside another function `f`

There are different ways of thinking about this:

- each call to `f` creates a 'different' function `g`
- the function `g` 'remembers' the local variables of `f`
- after a function call to `f` returns, its body is re-entered when a callback to `g` happens
- an inner function is some fixed code `g` plus a ***closure*** which stores values for its 'free variables'



Further callbacks

When we called `copy("...", "...")`, suppose we wanted to do something afterwards, involving the second file

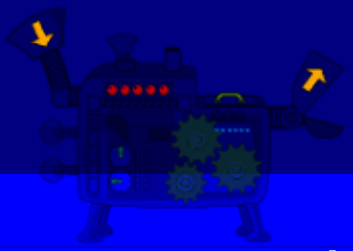
Then we would have to define `copy` as taking three arguments, the third being a callback

In general, any function which may return with work still outstanding, for whatever reason, has to be defined with a callback argument

Handling errors


```
"use strict";
var fs = require("fs");

function copy(file1, file2, done) {
  fs.readFile(file1, "utf8", ready);
  function ready(err, text) {
    if (err) done(err);
    else fs.writeFile(file2, text, finish);
  }
  function finish(err) {
    done(err);
  }
}
copy("in.txt", "out.txt", done);
function done(err) { console.log(err); }
```



Errors

A simple approach to handling an error `err` passed to a callback is to write `if (err) throw err;`

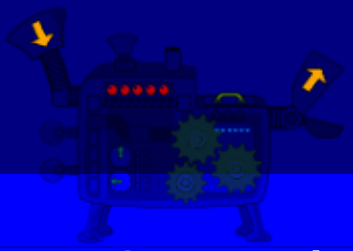
Some hard errors cause exceptions anyway, but for soft errors, this makes sure you get to see an error message

Alternatively, define your function with its own callback, and pass the error to the callback

Gathering files


```
// Join text from multiple files
"use strict";
var fs = require("fs");

function gather(files, gathered) {
  var all = "";
  gatherFile();
  function gatherFile() {
    var file = files[0];
    files.shift();
    fs.readFile(file, "utf8", ready);
  }
  function ready(err, text) {
    if (err) throw err;
    all = all + text;
    // Note files == [] wouldn't work
    if (files.length == 0) gathered(all);
    else gatherFile();
```



Loops

If you have a 'loop' of things to do involving I/O, you can't just write a normal loop

In the loop, you would call something involving a callback, and then after that you would need to wait somehow before starting the next iteration – or to put it another way, the callback function has no way of 'getting back inside the loop'

The only time you can write a normal loop is if you are going to start off a bunch of concurrent I/O activities

Concurrent file loading?


```
// Concurrently load text from files
"use strict";
var fs = require("fs");

function load(files, loaded) {
  var all = [];
  var count = 0;
  for (var i in files) {
    fs.readFile(files[i], "utf8", ready);
  }
  function ready(err, text) {
    all[i] = text;
    ...
// There is a problem with the variable i
// which can't be shared between the operations
```

Concurrent file loading


```
// Concurrently load text from files
"use strict";
var fs = require("fs");

function load(files, loaded) {
  var all = [];
  var count = 0;
  for (var i in files) loadFile(i);
  function loadFile(i) {
    fs.readFile(files[i], "utf8", ready);
    function ready(err, text) {
      if (err) throw err;
      all[i] = text;
      count++;
      if (count == files.length) {
        loaded(all.join(""));
      }
    }
```

Concurrent I/O

The loop completes straight away, leaving three outstanding file reads

The `ready` function needs to know what the index `i` is, in order to put the result in the right place in the output array, so it needs to be inside the `readFile(i)` function

The three ready calls happen in any order, filling in independent output slots, and a count allows the last one to detect when to call the callback