Node

Node is the best way to learn JavaScript before using it in web sites or web servers

node  objects  functions  I/O
Node

Node is a project at node.js, based on Chrome's V8 JavaScript engine

It provides a way of learning the language or developing JavaScript code away from browsers

It supports development of applications and web servers

If you find JavaScript difficult or irritating, consider using TypeScript or CoffeeScript or LiveScript, all of which compile into JavaScript
The name *JavaScript* is terrible, because it has little to do with Java and because it is a proprietary name – see [wikipedia entry](https://en.wikipedia.org/wiki/JavaScript).

The name used in IE browsers is *JScript* – IE copied the technology, but used a different name, so JavaScript and JScript are (almost) identical.

The standard name is *ECMAScript*, but we will do what everybody else does and call it JavaScript.

Version: we will concentrate on ES5 and ES6, but ES7 = ES2016, ES8 = ES2017 are becoming usable.
In this chapter, we are going to concentrate on learning the JavaScript language using Node.

In the next chapter, we will learn about using JavaScript on the client side, in browsers, on web pages.

Later, in the server side part of the course, we will look at using Node to build servers.

When you install Node, you get a command node (or nodejs or js) and a command npm (node package manager) for loading non-standard packages.
A scripting language is a "simple" programming language for small specific jobs, and traditionally it is:

+ aimed at non-programmers
- limiting for programmers
+ easy to learn
- lacking in features
+ targeted to one application area
- not truly general purpose
+ unfussy with few type limitations
- lacking in compile-time checking
+ interpreted, not compiled (but note V8)
- too inefficient for heavy use
There have been some recent important changes

Starting with Chrome's V8 engine, and spreading to other engines and languages, a system has been developed which (a) reconstructs the types which scripting languages don't have and (b) compiles just-in-time to machine code

So JavaScript is now *fast*

There have also been significant improvements in the language
Resources


Quick syntax reference: [wikipedia](https://en.wikipedia.org/wiki/JavaScript_syntax) and Node library reference: [nodejs.org/api/](https://nodejs.org/api/)

The language standards: **ES5**, **ES6**, **ES7**, **ES8**

For cleaning [JSLint](https://www.jslint.com/) and testing [QUnit](https://qunitjs.com/) and [platform tips](https://nodejs.org/api/

There are lots of other references you can Google but **beware** strict mode, browser or server bias, library modules (different in node/browsers/servers)
Modern ideas

As well as traditional ideas, these slides mention some modern ideas (some advocated by Douglas Crockford) which avoid JavaScript's bad parts:

- Strict mode
- Power constructors
- Different ways to do inheritance
- New features in ES6 (classes)
- New features in ES7 (async functions)
In sloppy (old) JavaScript, if you mis-spell a variable name, a new global variable is created for you.

JavaScript has lots of bad features, which can't be removed directly for backwards compatibility reasons.
JavaScript's strict mode can be used everywhere; in JavaScript 5+ systems, the compiler tells you if you misspell a variable.
Strict mode

Put "use strict"; at the top of every Node module.

In old JS, it is an "expression statement" which does nothing, but in JS5+, it switches on strict mode.

This mode disables some old, bad features of JS:

- Creating a global variable by mis-spelling
- Creating a global variable via an unset `this`
- Confusion from clashing argument or field names
- Inability to restrict access
- The unsafe `with` statement
- The unsafe `eval` function
Everything is an object, including strings, numbers, booleans, arrays, dates, patterns, and functions

An object has named fields that refer to other objects, it is like a map or hash table

Semicolons are optional, variables are lifted, objects can be recursive, 🔄 semicolons 🔄 lifting 🔄 data recursion
Here are two reasons to use semicolons:

- The rules for leaving them out are subtle, and it is easy to make mistakes - you don't really gain
- Almost all programmer's editors need the semicolons to handle indenting automatically - otherwise they make a mess

If you want to leave them out, a good general rule is to make sure the first line is clearly incomplete (e.g. ends with a binary operator)

```javascript
x =                (obviously incomplete) js
y + 3;           (continuation)
```
A JavaScript program is compiled in two phases

In the first, all the variables are defined, and in the second, the code is translated

That means you can make a forward reference to a later variable's *name* but not to the variable's *value* which hasn't been calculated yet

It is as if the declarations of the variable names (not definitions), and the definitions of named functions, have been *lifted* or *hoisted* to the top of the program or function
Recursive objects

```javascript
var house = { where: "house", out: garden };  // bad js
var garden = { where: "garden", in: house };  
```

This could be part of an adventure game, but field `house.out` is `undefined` because the code which initialises `garden` hasn't been executed yet, so a correction is:

```javascript
var house = { where: "house" };  
var garden = { where: "garden", in: house };  
house.out = garden;  
```

Now the two objects refer to each other
var circle = { x: 3, y: 4, radius: 5 };  
var square = { x: circle.x, y: circle['y'] };  
circle.diameter = 12;  
circle.radius = 6;  
square['y'] = 7;

At any time, you can access, add, change, delete a field using a static or dynamic style

The static style `o.x` needs the field name (a) to have identifier format and (b) to be known at compile time

In the dynamic style `o['x']` or `o[v]` it is a string value which is used as the field name

arrays
Arrays and fields

Writing `obj["1"]` to look up a field of `obj` called `1` is the same thing as writing `obj[1]` to look up entry `1` in an array `obj`.

An array is an object with fields called `0`, `1` etc.

And it is also a hash map from string keys "0", "1" etc. to values.
There is no type checking at compile time, but a lot at run time ("does object have field x or function f?")

There is a `typeof` operator which is nearly useless - it picks out primitives `string`, `number`, `boolean`, `function`, plus `object` and `undefined`

It returns `object` for an array, or for any types you define for yourself, but works a bit better in browsers for web objects

Later, we will look closer at types (actually prototypes)
Strings

Strings are immutable (you can't change them); a character is a length-one string \(\textit{characters}\); you can use single or double quotes, e.g. "don't" or 'say "Hi"'; to change a string, write \(s = s.toUpperCase()\).
Characters

JavaScript, like Java, was designed in the days when people didn't expect Unicode codes to rise above 65535.

Strings are stored in UTF16, that is with two bytes per character, and a pair of characters is used for each code value beyond the two-byte range.

Most recent languages opt for UTF-8 to represent strings.
"use strict";
var i = 42;
var f = 42.56789;
console.log(typeof i, typeof f,
    Math.floor(f), i.toFixed(2), f.toFixed(2));

> node numbers.js
number number 42 42.00 42.57

There is only one number type (double precision),
integers are \texttt{xxx\cdot0}, \texttt{/} is exact division, \texttt{%} works

Numbers are immutable (as you would expect from
other languages, even though numbers are objects)
Arrays are not really a separate type

a == [] gives false (two different empty-array objects) so a.length == 0 works better

Check out functions push, pop, shift, unshift, join, slice, splice, concat, sort
Typed Arrays

Although JavaScript has no integers, it does have arrays of integers

See **typed arrays**, but check [Can I Use](https://caniuse.com) first

Arrays of bytes, two-byte integers, 4-byte integers, all signed and unsigned, are available when efficiency or compactness are important
"use strict";
console.log(typeof true);
console.log(0 ? true : false);
console.log(0 == false, 0 === false);

There is a boolean type with immutable constants `true` and `false`.

But 0, ",", NaN, null, undefined are treated like false and everything else like true. 

The `==` operator is lenient, `===` is stricter & safer.
Feature tests

The lenient nature of `if (test)` can cause problems

However, it is quite handy to test for fields or functions:

```javascript
if (object.field) ...
```

If the field exists, it is treated like `true` and if not `(undefined)` it is treated like `false`, so this tests whether the field `exists`
Errors

If you try to access something, there are three kinds of error response you might get from JS:

- an exception, i.e. a program crash
- the `null` value, a non-existent pointer
- the `undefined` value

It is not intuitive which circumstances produce which of these three responses, you have to rely on testing and experience, but `null == undefined` helps.
Statements such as `if`, `switch`, `while`, `for`, and compound statements with curly brackets, are almost identical to other C-family languages, but local variables are different, e.g.: 🔄 C family.

```
"use strict";
function ... {
    for (var i=0; i<n; i++) {
        ...
        for (var i=0; i<m; i++) { ... }
    }
}
```

Since all variables are lifted to the start of the function, the two `i`'s are the same variable (ES6 has `let`).
The C family

There are 50+ languages related to C, including C++, Java, JavaScript, ...

They can usually be recognized by the curly brackets and semicolons

They all have if statements, while loops, for loops, switch statements and so on
"use strict";
function show(s) { console.log(s) }
show("Hi");
show();
show(1, 2);
show(show);

A function has no fixed return type, no fixed arg types, and not even a fixed number of args

A function is an object, and can be passed around and printed out
**Function notation**

```javascript
function twice(n) {
    return n+n;
}
```

An alternative is (note semicolon needed for editors)

```javascript
var twice = function (n) {
    return n+n;
};
```

Some people write this *(unreadable garbage)*:

```javascript
console.log(function (n) {
    return n+n;
});
```
callbacks (see later) are often written like this:

```javascript
do_some_io(..., function (arg) {
    ...
});
```

which is terrible; it has a vertical `{ }` block inside a horizontal `( )` function call; it should be

```javascript
do_some_io(..., ready);
function ready(arg) {
    ...
}
```

callbacks are difficult enough as it is
"use strict";
var object = { name: "me" };
object.show = display;
function display() { console.log(this.name); }
object.show();

A *method* is a function attached as a field 

A call `object.show()` , as well as finding a field and calling it, sets `this` to the object 

If you call `display()` directly, `this` is undefined (strict mode) or the global object (sloppy mode)
In the example, the method name `show` used to attach the function to the object is different from the function name `display`, for clarity.

However, it is quite normal to give them the same name.

There is no clash, because one is a field name inside the object, and the other is a global name.
A method call `object.show()` can't be split into the two steps `f = object.show; f()` because `this` would then be undefined.

There is another way of making the call, which is `show.call(object)`, useful in dynamic situations, or if the method isn't actually attached to the object.

Every function has a `call` method, where `this` is set to the first argument, and the function is called with the remaining arguments passed to it.
Here's my favourite way of writing a constructor, and therefore a pseudo-class:

```javascript
function newPet(name) {
    var age = 0;
    return { show, older };,
    function show() { console.log(name, age); }
    function older() { age++; }
}
var garfield = newPet("Garfield");
```

- naming, no this(+), private fields(+), easy events(+), awkward inheritance(-), awkward testing(-), memory(-).
A *power constructor* is a plain function which creates an object, sets up its fields and methods and returns it.

In this unit, the name starts with `new`, e.g. `newPet`, as a reminder that it is a constructor (and to make it less likely to type `new Pet(...)` by mistake.

The object `{name, show}` is short for:

```
{ name: name, show: show }
```

The variable names are used as the field names.
In `newPet`, the keyword `this` isn't used anywhere.

Just as in Java, for example, the fields are accessed directly from the methods, without any `this`. prefix.

The more official constructor techniques (classical constructors and ES6 class constructors) mean that all methods irritatingly have to use the `this`. prefix to access the fields all the time.
Private fields

The argument `name` and the variable `age` are local to the function, and are remembered as private fields of the object when the function returns.
In a browser, you could write `setInterval(older, 5000)` to call the `older` method regularly.

This works, with the method able to access the private fields of its object directly.

With other types of constructor, you have to use global variables or an inner function with a closure, to give the method access to the object.
Awkward inheritance

Inheritance is arguably more awkward than with other types of constructor, because an extended object doesn't have access to the private fields of the original object.

**But:** inheritance is overrated because of the close coupling, and is unsafe if you can break the promises of the original object, and isn't needed nearly as often as people think.
Awkward testing

Debugging can be awkward with power constructors, because you can't test or print out the private fields from outside the constructor.

**But:** if you do proper automated unit testing inside the constructor, all is well.
With the power constructor scheme, there is another hidden object (closure) for each visible object, using up memory.

**But:** the situations where memory is a problem are rare, and other schemes may use more memory than you think.
Disadvantages of power constructors are:

- they don't support inheritance, because inheriting objects can't get at the private fields (but inheritance is overrated)
- the private fields are not easily printed from outside for debugging (but autotesting is better)
- when there are millions of objects, memory usage is unnecessarily high (but special techniques such as typed arrays are needed then anyway)
This traditional approach has *hardly any advantages*

```javascript
function Pet(name) {
  this.age = 0;
  this.name = name;
  this.show = show;
  this.older = older;
  function show() {
    console.log(this.name, this.age);
  }
  function older() {
    this.age++;
  }
}
var dory = new Pet("Dory");
```

- `new(-)`
- `this(-)`
- `Java(-)`
- `confusing inheritance(-)`
The **new** keyword creates a blank object and sets it as this before calling the constructor function: roughly

```javascript
function new(con, args) {
    var x = {};
    x.con(args);
    return x;
}
```

That means you have no control inside the constructor over how the object gets created.
The `this` keyword

Every time you access a field, you have to use `this` as a prefix.
Java compatibility

This style was designed to look like Java

But it is artificial, confusing, and adds nothing, because JavaScript isn't actually much like Java at all

It would be much better to develop JavaScript in its own direction
In this style, a prototype is attached to the constructor:

```javascript
var animal = ...;
Pet.prototype = animal;
```

The `new` keyword then gets the prototype from the constructor and attaches it to the object automatically.

The `prototype` field is attached to `Pet` (weird) and isn't `Pet`'s prototype (weird).

Surely a direct, controlled, visible approach is better:

```javascript
var pet = Object.create(animal);
```
This ES6 approach has *hardly any advantages*

```javascript
class Pet {
    constructor(name) {
        this.name = name;
        this.age = 0;
    }
    show() { console.log(this.name, this.age); }
    older() { this.age++; }
}
var tweety = new Pet("Tweety");
```

*new(·), this(·), Java(·), inheritance(±)
The class approach still uses the `new` keyword, which means you still don't have any control over how the new object is created.
The *this* keyword

Every time you access a field, you still have to use *this* as a prefix.
This style looks even more like Java

But it is still artificial and confusing
Some things like inheritance do seem a bit more natural

So the advice is: *if* you don't like the power constructor approach, and you have been using the classical approach, try out the class approach, check its availability in the browsers you care about, and consider upgrading to it.
Constructor advice

I recommend that you *either* use power constructors all the time (because of their advantages) *or* use ES6 classes all the time (because of their Java familiarity or because they are standard)

There don't seem to be any good reasons to use 'classical constructors'  

It is a matter of taste, though
Closures

Closures are important because they are used for event handling in JavaScript, appearing in many JS tutorials.

Suppose a method `m` mentions `x`:

```javascript
m() { ... x ... }
```

The method is 'open' because `x` is outside it somewhere, so `m` needs to be 'closed' by providing `x`.

A closure is an object which closes `m` by providing `x`.

It is an implicit object, not `this.x`.
"use strict";
function createCount() {
  var n = 0;
  return count;
  function count() { console.log("n =", ++n); }
}
var count = createCount();
count(); count(); count();

When an inner function is defined, it can access the args and local variables of the outer function.

This works even when the outer function returns.

A closure object is created to contain them, with each outer call creating a new closure object.
Importance of closures

Many programmers say "closures are the best feature"

It is true that they are essential, and it is important to understand them when it comes to event handling, i.e. input/output and concurrency

However, closures are a historical legacy left over from before object oriented programming

A method is associated with two 'special' objects: (a) its owner (i.e. the one it is attached to) and (b) its closure, and they 'ought' to be the same

*comparison*
Owner and closure

The owner of a method is explicit, visible, controllable, and understandable, but its fields have to be accessed using a 'this.' prefix.

The closure of a method is implicit, invisible, uncontrollable and (experience suggests) poorly understood, yet its fields are directly accessible.

Surely the design of the language would be much better if they were the same object, both explicit and convenient.
How closures work

The methods for one object created by the constructor all have the same closure attached.

The closure is where the variables live, which is what makes them private.
... var car = Object.create(vehicle);

Each object has a *hidden* field pointing to a prototype object which defines default methods (or fields)

In a field/method lookup `o.x` on object `o` with prototype `p`, if `x` isn't found attached to `o`, `p.x` is used

This is useful to share lots of methods between lots of objects, to save space (or improve design) 🔄 create
The function `Object.create` was new in JavaScript 5, so isn't present in old browsers but it can be defined as a polyfill

```javascript
if (! Object.create) {
  Object.create = create;
  function create(proto) {
    function F() { }
    F.prototype = proto;
    return new F();
  }
}
```
We can use prototype inheritance to share methods:

```javascript
function newAnimal() {
    return { show, older }
    function show() { console.log(this.name, this.age); }
    function older() { this.age++; }
}

var animal = newAnimal();
function newPet(name) {
    var p = Object.create(animal);
    p.name = name;
    p.age = 0;
    return p;
}

var minnie = newPet("Minnie");
```

- changes
- sharing
Changes

The example still uses power constructors for maximum control.

And it uses `Object.create(prototype)` for maximum clarity.

But it switches to public fields attached to the object and accessed with `this` to allow inheritance.
Sharing

When objects are created via a prototype, they can share methods, reducing the size of the objects.

Fields, however, (except for constants) need to be attached to each object, because they differ from object to object.

So a method attached to the prototype must access fields attached to a newly created object.

That's easy if the fields are public, using `this`, but very difficult otherwise.
ES6 classes make inheritance (arguably) more readable:

```javascript
class Animal {
    constructor(name) {
        this.name = name;
        this.age = 0;
    }
}

class Pet extends Animal {
    constructor(name) {
        super(name);
    }
    show() {
        console.log(this.name, this.age);
    }
    older() {
        this.age++;
    }
}

var lassie = new Pet("Lassie");
```

There are keywords `class`, `extends`, `constructor`, `super` (and polyfills for older browsers)
Modules have been difficult for a long time, with competing non-standard approaches in node (`require`) and in browser support libraries (AMD, CommonJS).

Modules were finally standardized in ES6.

Is it now possible to develop and test non-IO modules in node, and then use them without change on a web site?

Just about!
• design most modules as non-IO, for testing
• put modules in files ending with .mjs
• export a class or power-constructor from each
• write: import xxx from "./xxx.mjs";
• node --experimental-modules xxx.mjs
• add the .mjs extension to your web server
• <script src="xxx.mjs" type="module"... 
• for Chrome, deliver pages as text/html
• for Firefox, visit about:config, switch on dom.moduleScripts.enabled
Power modules

```javascript
export default function newPet(name) {
    var age = 0;
    return { show, older }
    function show() { console.log(name, age); }
    function older() { age++; }
}
```

Defines a pseudo-class as a power constructor

```javascript
import newPet from './pet.mjs';
var g = newPet("Garfield");
```

Global variables and functions are private unless exported, "use strict" is the default, and a browser runs a module after the page is ready.
Class modules

```javascript
export default class Pet {
    constructor(name) {
        this.name = name;
        this.age = 0;
    }
    show() { console.log(this.name, this.age); }
    older() { this.age++; }
}

Defines a pseudo-class as a power constructor

import Pet from "./pet.mjs";
var tweety = new Pet("Tweety");
```
Input and output libraries are *different* in Node from in browsers, but the *style* is the same.

So we can still use Node to investigate *asynchronous* calls, *events* and *callbacks* which are the same as in browsers.

So far, we have used just `console.log` for writing, which *is* available in browsers, for debugging, now we investigate event-driven input/output.
"use strict";

process.stdin.setEncoding('utf8');
process.stdin.on('data', receive);

function receive(text) {
    console.log("You typed", text);
}

process.stdin doesn't exist in browsers

setEncoding specifies text instead of binary

The program repeatedly waits for you to type in a line, then echoes it
This line specifies that the program should now process each event of type 'data' from stdin, which is when a line has been typed and is ready.

The program will now not stop until no more events are possible - it is event-driven or in an event loop.

The second argument is a callback, a function that will be called when an event happens - it is passed as an object, not called yet.
"use strict";
var fs = require("fs");

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

function ready(err, text) {
  console.log(text);
}

fs is the file system module

This way of loading modules is Node-only (pre ES6)

readFile reads a whole file, with a callback

Almost all I/O in Node uses callbacks for concurrency

ℹ️ concurrency  ⚙️ threads
Concurrency usually has threads sharing data (e.g. Java) or threads communicating using channels (e.g. Go).

Alternatively, Node uses the event loop for highly concurrent servers, arguably easier to program with.

The programmer chops up the program into callback functions, and Node executes calls with a queue, so that lots of things can be going on at once.

This keeps one processor busy when there are things to do, but can't take advantage of multiple processors, so on a multi-core processor, several identical Node servers are needed.
Node's single-threaded approach using asynchronous calls and events and callbacks is confusing.

In languages with threads, the code is much more natural.

But the simplicity of threads is an illusion; see Threads Are Evil.

You have to balance that against: Callback hell.
"use strict";
var fs = require("fs");

copy("in.txt", "out.txt");

function copy(file1, file2) {
    fs.readFile(file1, "utf8", ready);
    function ready(err, text) {
        copy2(file2, err, text);
    }
}

function copy2(file2, err, text) {
    if (err) throw err;
    fs.writeFile(file2, text, end);
}

function end(err) {
    if (err) throw err;
    console.log("Done");
}
One copying operation consists of three function calls, one to each of these functions

`copy` and `copy2` each end with an I/O call, which returns immediately, `without` having carried out the I/O operation yet

This is an asynchronous call, designed to allow the program to carry on doing something else while it is waiting
The event loop

copy("in.txt", "out.txt");

When this call is made, it returns immediately, without having done the copy (and if you want to do something after it, you add a callback to copy)

Instead, the copy operations have been "queued up", waiting for events to happen to indicate completion

After this call, the program reaches the end

Normally, the program would stop, but if events are still possible, it goes into an event processing loop instead
When ready is called, it is passed two arguments err and text - you can find this out in the documentation.

But copy2 needs file2 - how do we pass that?

The approach taken here is to write a one-line callback function inside copy to add the extra argument (this is an implicit use of closures).
A common alternative is to nest `copy2` inside `copy`, possibly making `copy2` anonymous.

This quickly leads to **callback hell**.
A less common alternative is to use the `bind` method attached to every function

Bind creates a new function in which the first argument(s) to `copy2` have been filled in, leaving the two standard arguments

```javascript
f.bind(o,x...) is the same as
function(y...) { o.f(x...,y...); }
```
Another uncommon alternative is to use the "arrow" or "lambda" notation for small anonymous functions.
In browser scripts, Node's library modules like `fs` are not available (though globals `console`, `Math`, `JSON` are) but instead, there are web-page objects and methods, and simple callback-style event handling.

In servers, Node's library modules are available, including `fs`, `http`, `https`, and you can add a database module, and handling each request involves many small function calls, chained with callbacks.