Introduction to Nodejs/ES6
Part2 Angular2+ (breaking down complexities)

By George Franklin

Our Sponsors

Bank of Ireland JET BRAINS redgate

Limerick AI – (Applications and Games) Software Development MeetUp

http://meetup.com/LimerickAiSD

Twitter @LimerickAiSD

Big Thanks To

- For Everyone Joining & Attending

- Pat Carroll BOI
- Deirdre Twomey BOI

- Brian Keating for his talk on Kotlin

- Are you interested in Machine Learning, Data science, Big Data, Statistics - Like Building developing Models With Python or R?
- Artificial Intelligence – Traditional AI? Search, optimization, Puzzles, NLP (Natural Language Processing), NLP(Neuro-linguistic programming)
- Like building Neural Networks i.e. Deep Learning? TensorFlow, Keras, PyTorch , Caffe, Torch, Theano, CNTK ,Chainer
- Games Design Development, Graphics 2d/3d/Modelling Blender.. etc
- General cross platform Software Development? Web,Mobile,Desktop
- IoT/Embedded System – Arduino, raspberry pi, Intel IoT Boards, Automation, Google Nest,
- Web Development? Angular, React,Vue.js, Html5,Css, Sass, WebPack, Javascript, Typescript
- Procedural/ object oriented - C, C++, C# Java, Python, Go, Kotlin, ObjectiveC
- Functional Languages F#, Scala, Haskell, Erlang, Clojure, Lisp
- Testing/QA – Testing Strategies / interested in Manual/Automated Testing
- Devops – Docker. Kubernetes, Puppet,Chef, Ansible ,Salt, Terraform
- Systems.Network Engineer – Network virtualization, Open Switch, NFV , Cisco, Vmware, openStack, Azure, Aws, Google
- This Meetup is for people interested in learning more about Artificial intelligence, Developing AI Applications or Games and General Software Development across different platforms e.g Web, Mobile, Desktop and IoT Devices. (Windows10 & IoT, Linux / MacOS /IOS / Android).

In Session 1

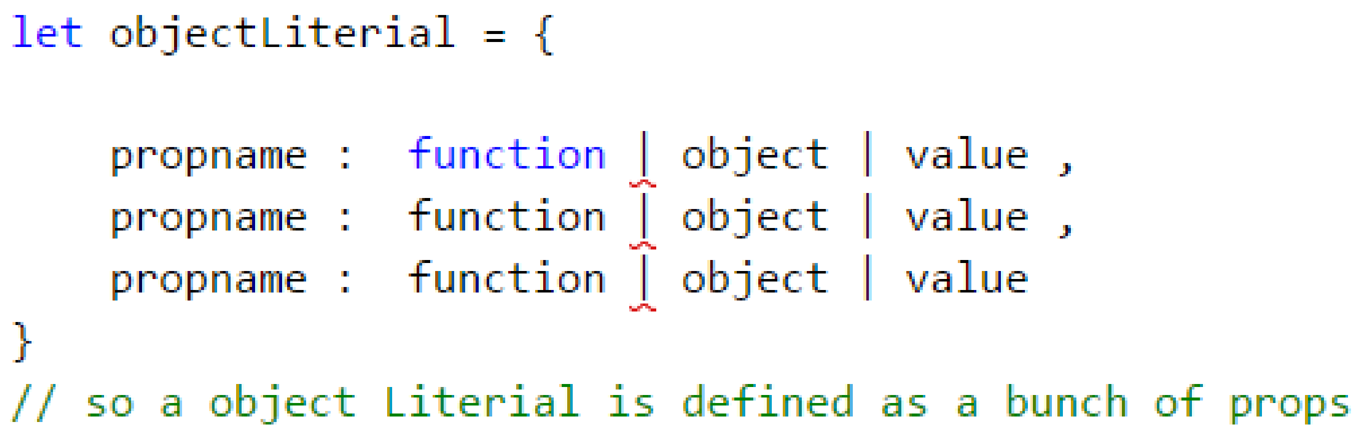
So before we talk Nodejs

- Just a few Javascript essentials so we are using the same terminology.
- The Javascript types
- The Javascript Object Literial – what it is?
- What it is not – i.e. It’s not JSON !!!
- Hoisting var , let and const.
- Function declarations are automatically raised. Var function expressions variables get hoisted, but are undefined. i.e. the function definition does not.
- We cover the other aspects of Javascript after discussing the internals of Nodejs

JavaScript Object Literal

- My Definition – The Javascript Object Literial is an object stored on the Heap, which contains a bunch of properties i.e. these props are the public interface for how you wish to access whatever you have assigned to them.

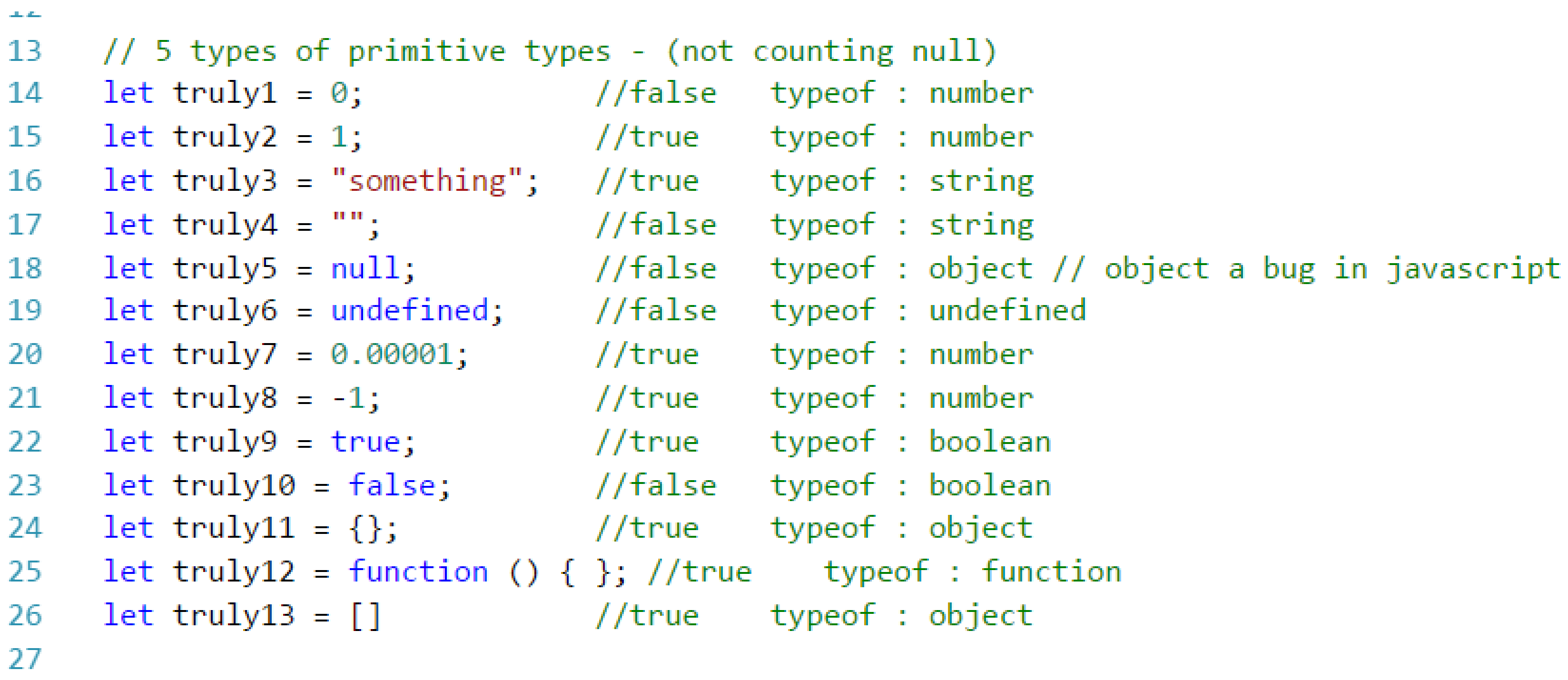
- Remember Javascript has no direct notion of private or public access modifiers. We define the properties that we wish the object to have and they become publicly available. ES6 introduced some shortcuts to defining properties, we will discuss them later.



```
let objectLiterial = {

    propname :  function | object | value ,
    propname :  function | object | value ,
    propname :  function | object | value
}
// so a object Literial is defined as a bunch of props
```

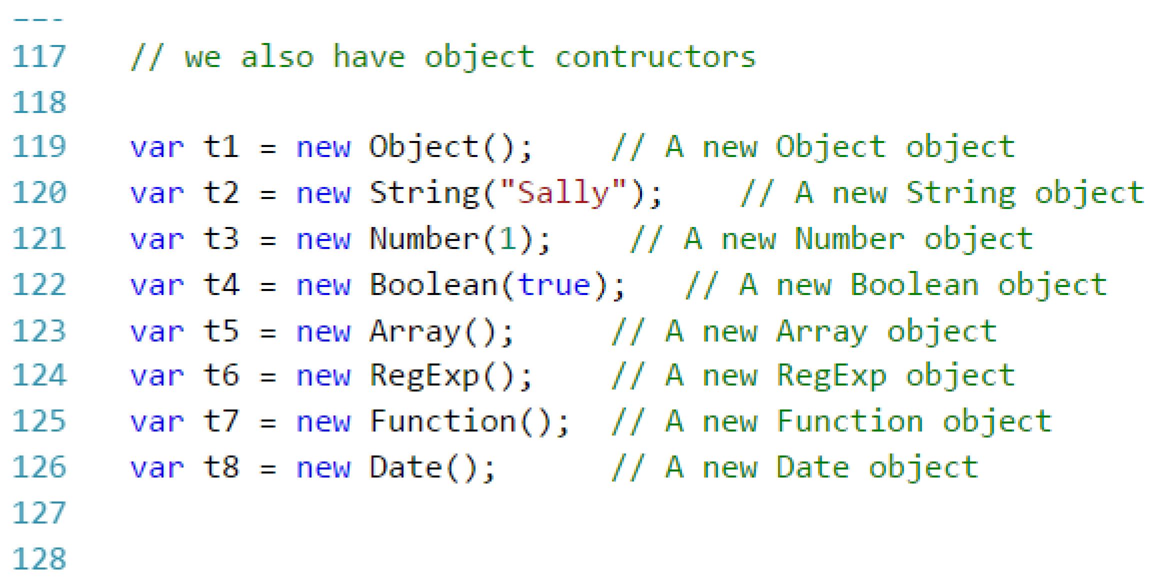
Not shown above – but we can also have arrays as props i.e.
let ob = { firstname: “sally” , myarray : [‘one’, ‘two’] }

Javascript Types

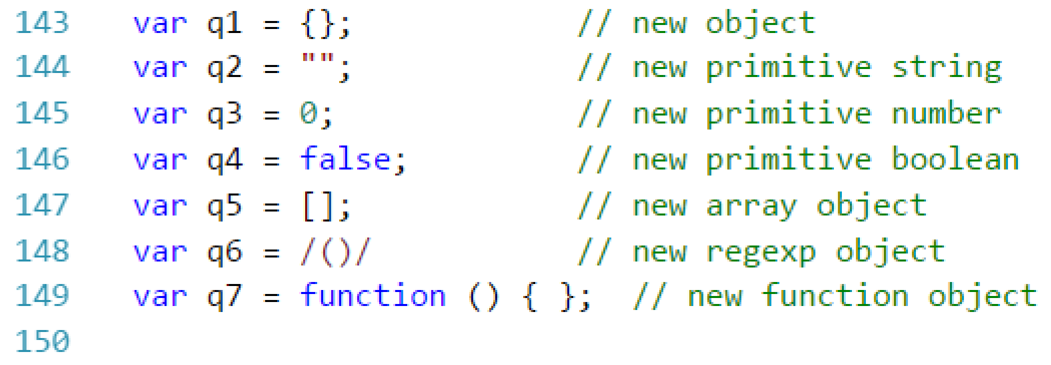


```
// 5 types of primitive types - (not counting null)
let truly1 = 0;              //false    typeof : number
let truly2 = 1;              //true     typeof : number
let truly3 = "something";    //true     typeof : string
let truly4 = "";             //false    typeof : string
let truly5 = null;           //false    typeof : object // object a bug in javascript
let truly6 = undefined;      //false    typeof : undefined
let truly7 = 0.00001;        //true     typeof : number
let truly8 = -1;             //true     typeof : number
let truly9 = true;           //true     typeof : boolean
let truly10 = false;         //false    typeof : boolean
let truly11 = {};            //true     typeof : object
let truly12 = function () { }; //true     typeof : function
let truly13 = []             //true     typeof : object
```

Using alternate constructors

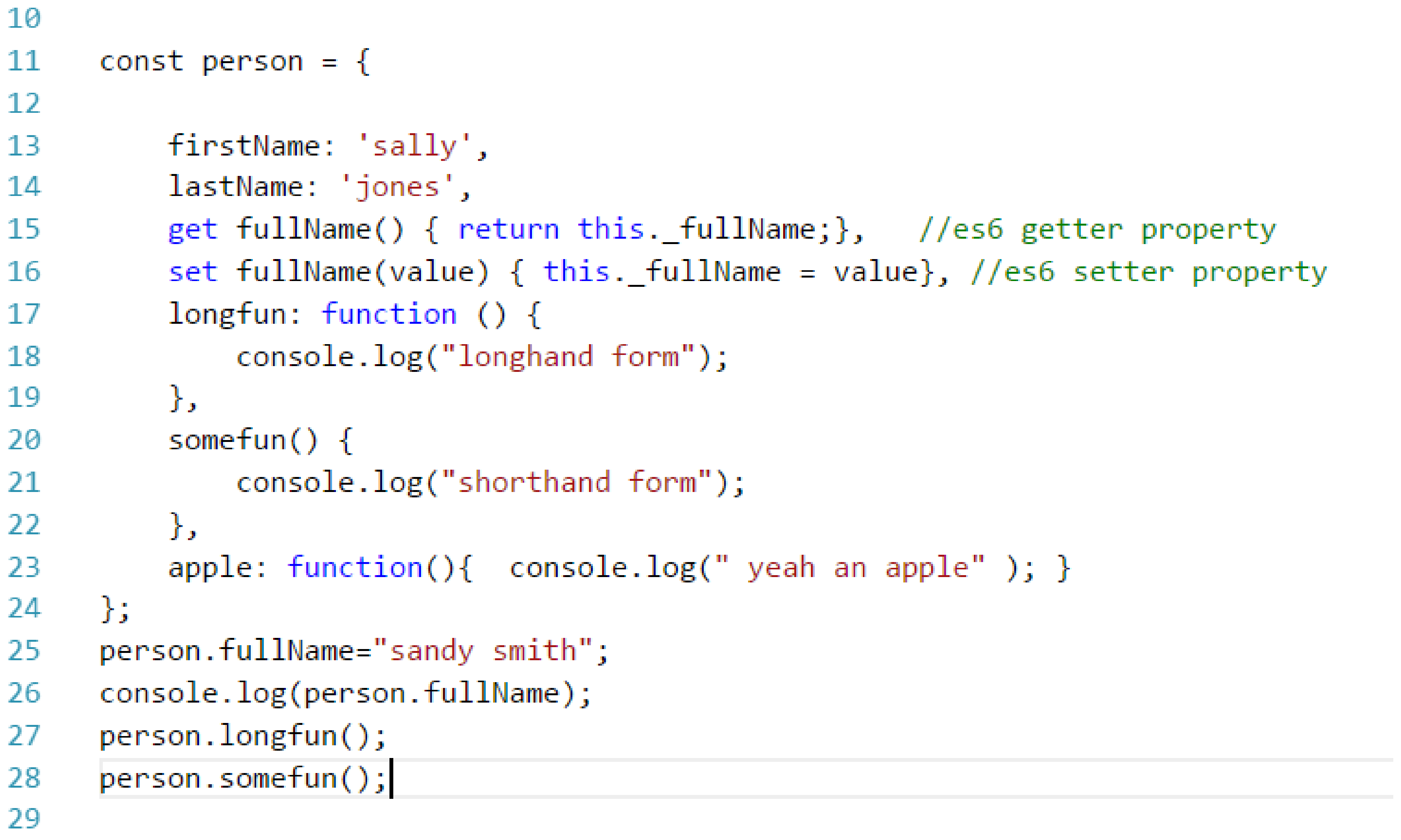


```
// we also have object contructors

var t1 = new Object();    // A new Object object
var t2 = new String("Sally");    // A new String object
var t3 = new Number(1);    // A new Number object
var t4 = new Boolean(true);   // A new Boolean object
var t5 = new Array();     // A new Array object
var t6 = new RegExp();    // A new RegExp object
var t7 = new Function();  // A new Function object
var t8 = new Date();      // A new Date object


```



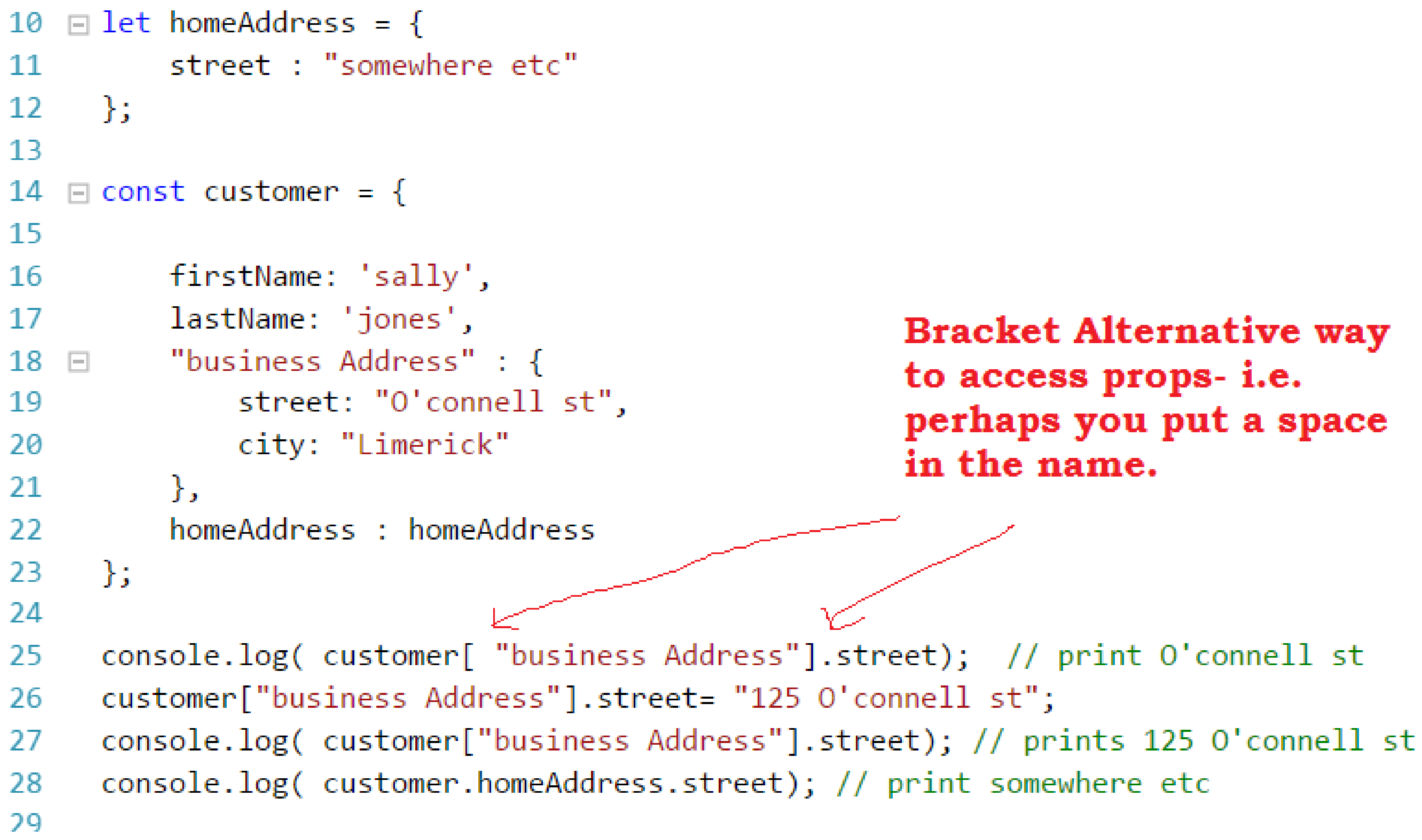
```
var q1 = {};                // new object
var q2 = "";                // new primitive string
var q3 = 0;                 // new primitive number
var q4 = false;             // new primitive boolean
var q5 = [];                // new array object
var q6 = /()/               // new regexp object
var q7 = function () { };  // new function object

```

- Now lets look at various properties types



```
const person = {

    firstName: 'sally',
    lastName: 'jones',
    get fullName() { return this._fullName;},   //es6 getter property
    set fullName(value) { this._fullName = value}, //es6 setter property
    longfun: function () {
        console.log("longhand form");
    },
    somefun() {
        console.log("shorthand form");
    },
    apple: function(){  console.log(" yeah an apple" ); }
};
person.fullName="sandy smith";
console.log(person.fullName);
person.longfun();
person.somefun();
```

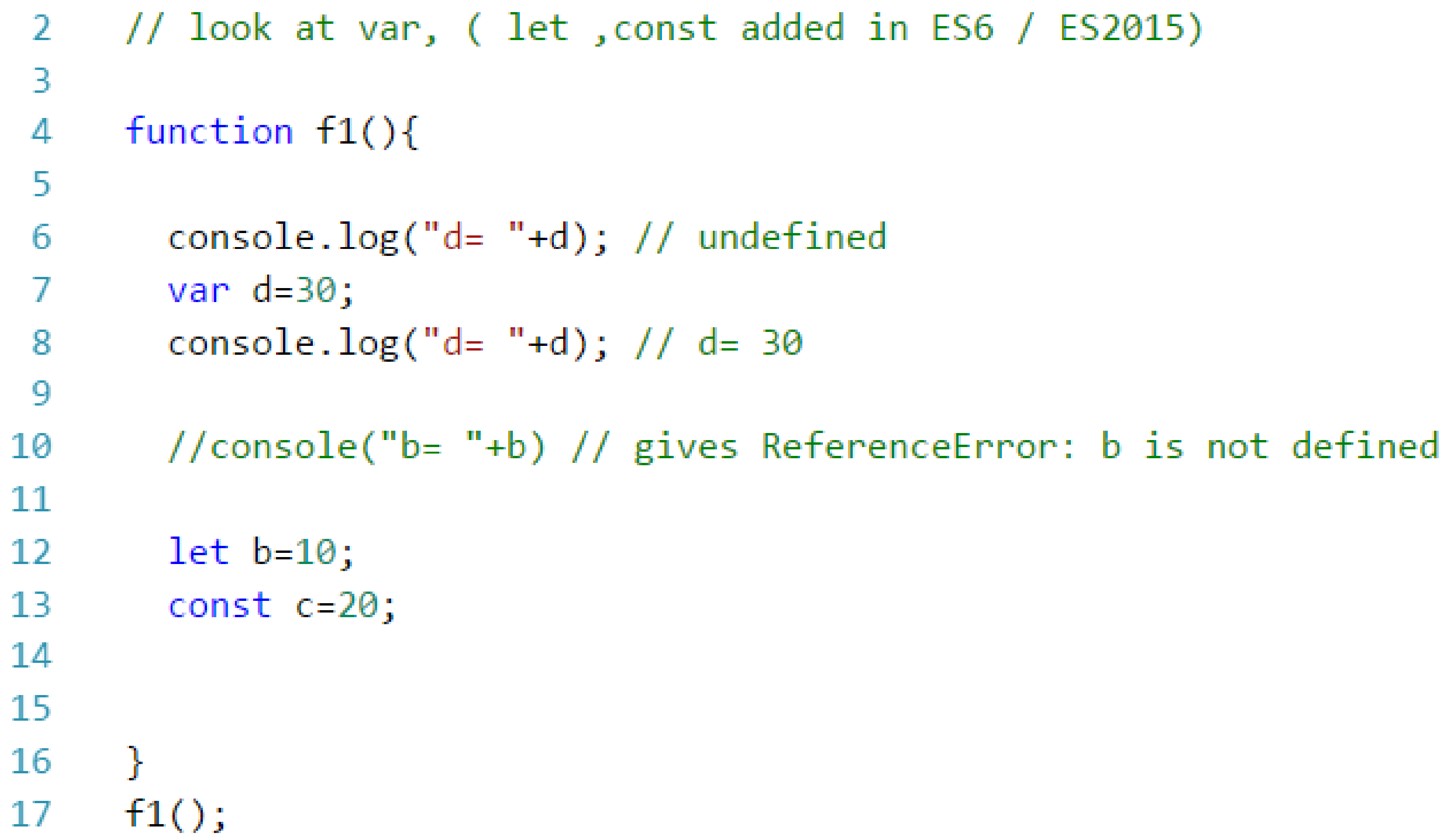
Accessing Props via the bracket syntax



```
let homeAddress = {
    street : "somewhere etc"
};

const customer = {

    firstName: 'sally',
    lastName: 'jones',
    "business Address" : {
        street: "O'connell st",
        city: "Limerick"
    },
    homeAddress : homeAddress
};

console.log( customer[ "business Address"].street);  // print O'connell st
customer["business Address"].street= "125 O'connell st";
console.log( customer["business Address"].street); // prints 125 O'connell st
console.log( customer.homeAddress.street); // print somewhere etc
```

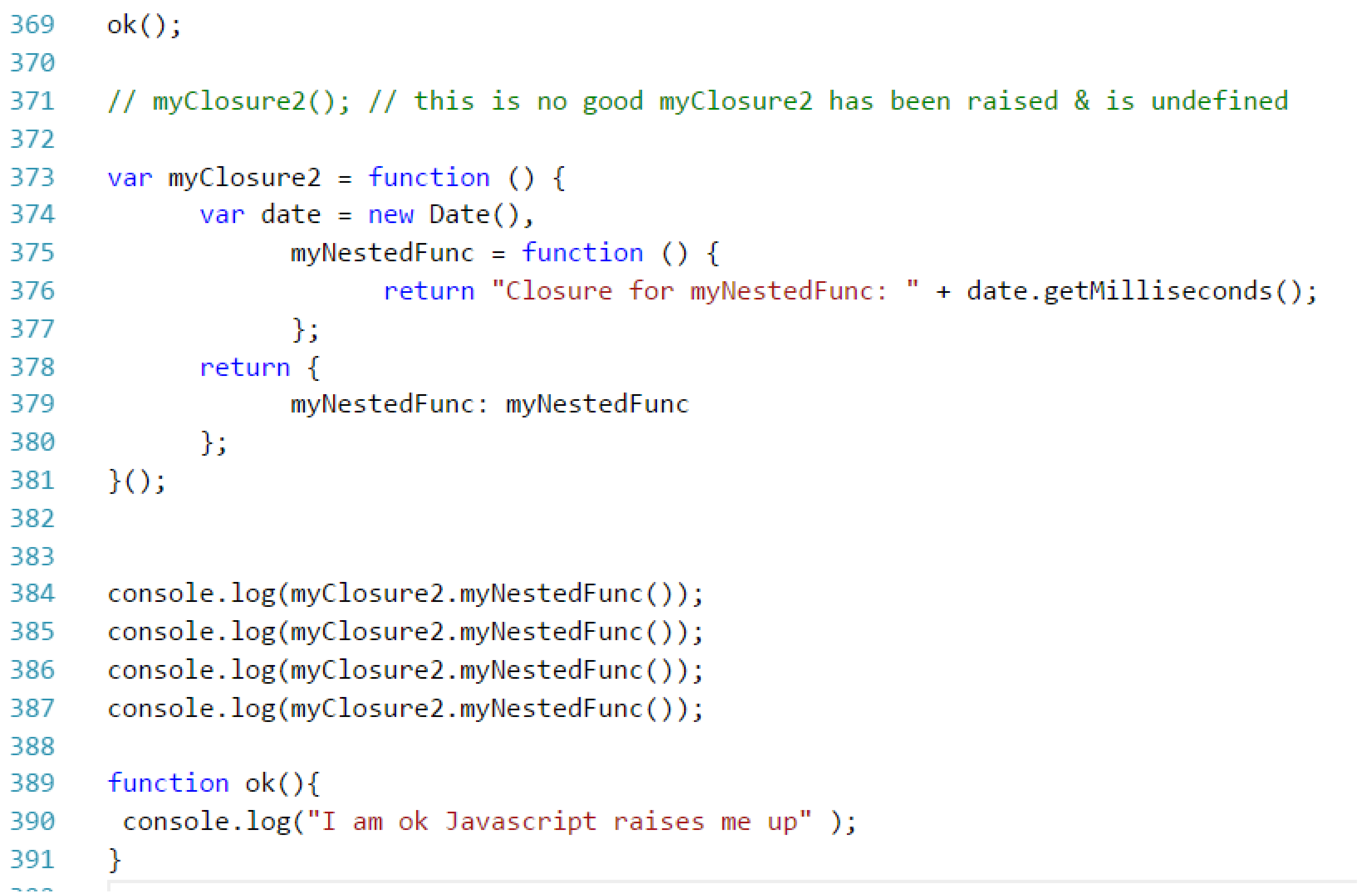
Bracket Alternative way to access props- i.e. perhaps you put a space in the name.

Var Hoisting



```
// look at var, ( let ,const added in ES6 / ES2015)

function f1(){

  console.log("d= "+d); // undefined
  var d=30;
  console.log("d= "+d); // d= 30

  //console("b= "+b) // gives ReferenceError: b is not defined

  let b=10;
  const c=20;


}
f1();
```

Raising/hoisting functions and closure for good measure



```
ok();

// myClosure2(); // this is no good myClosure2 has been raised & is undefined

var myClosure2 = function () {
      var date = new Date(),
            myNestedFunc = function () {
                  return "Closure for myNestedFunc: " + date.getMilliseconds();
            };
      return {
            myNestedFunc: myNestedFunc
      };
}();


console.log(myClosure2.myNestedFunc());
console.log(myClosure2.myNestedFunc());
console.log(myClosure2.myNestedFunc());
console.log(myClosure2.myNestedFunc());

function ok(){
 console.log("I am ok Javascript raises me up" );
}
```

To answer the confusion of Javascript Objects and JSON

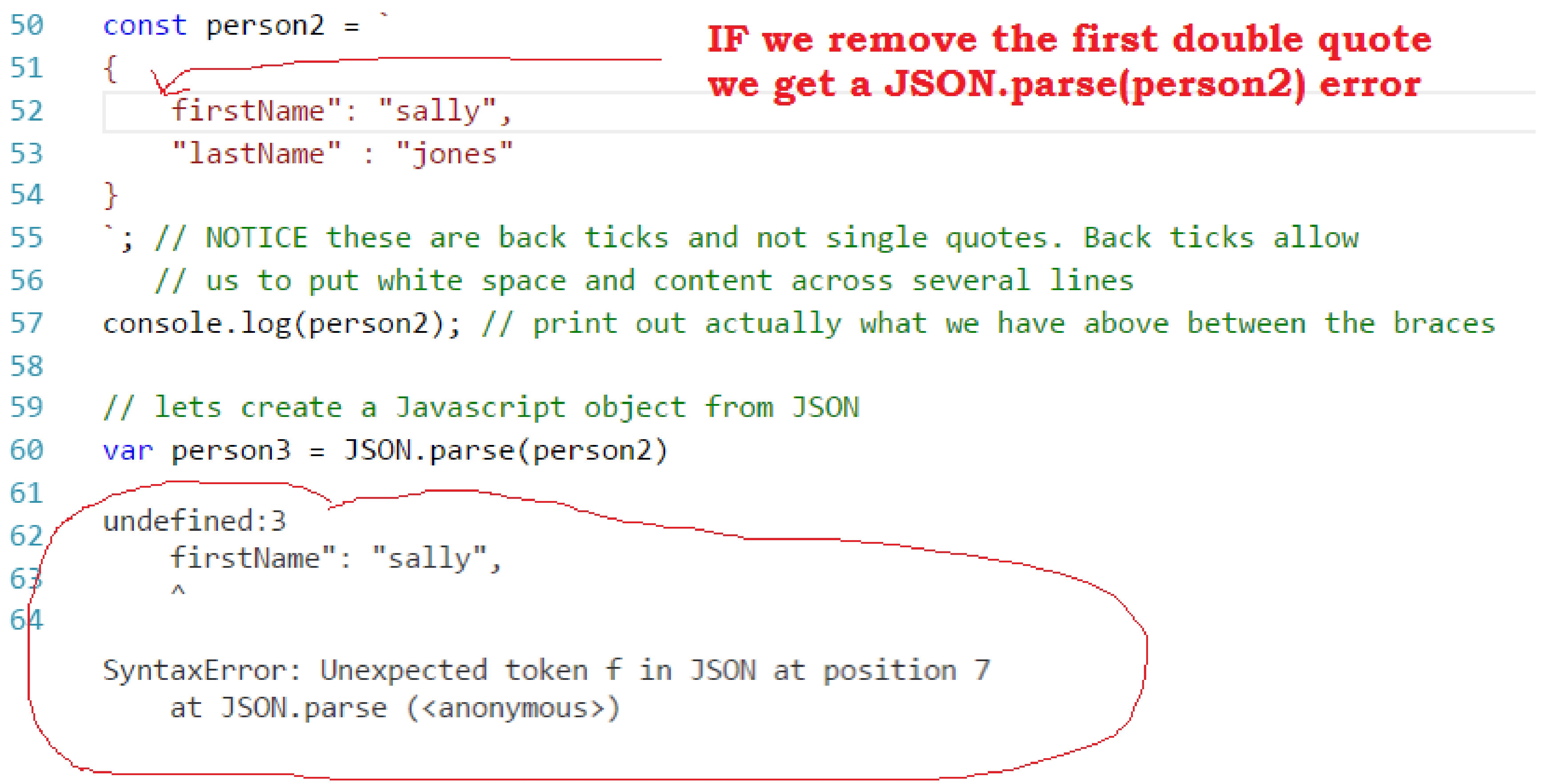
- JSON is is a data description language.http://www.json.org
- it is a "lightweight data-interchange format." - not a programming language.

- "basic types" supported are:
- Number (integer, real, or floating point)
- String (double-quoted Unicode with backslash escaping)
- Boolean (true and false)
- Array (an ordered sequence of values, comma-separated and enclosed in square brackets)
- Object (collection of key:value pairs, comma-separated and enclosed in curly braces)
- null

Example of parsing Json

```
// Notes about differences between JSON Data storage and
// an Javascript Object literal
// JSON properties must be quoted with double quotes, where javascript
// literials do not require this
// JSON string values must be quoted with double quotes;
// ( single quotes and template literials are not permmited )
// JSON does not support function properties
const person2 = `
{
    "firstName": "sally",
    "lastName" : "jones"
}
`; // NOTICE these are back ticks and not single quotes. Back ticks allow
   // us to put white space and content across several lines
console.log(person2); // print out actually what we have above between the braces

// lets create a Javascript object from JSON
var person3 = JSON.parse(person2)

console.log(person3); // prints out { firstName: 'sally', lastName: 'jones' }

console.log(JSON.stringify({ x: 5, y: 6 }));
// prints out {"x":5,"y":6}
```

But lets prove we need Valid JSON for us to turn it back into a Javascript Object

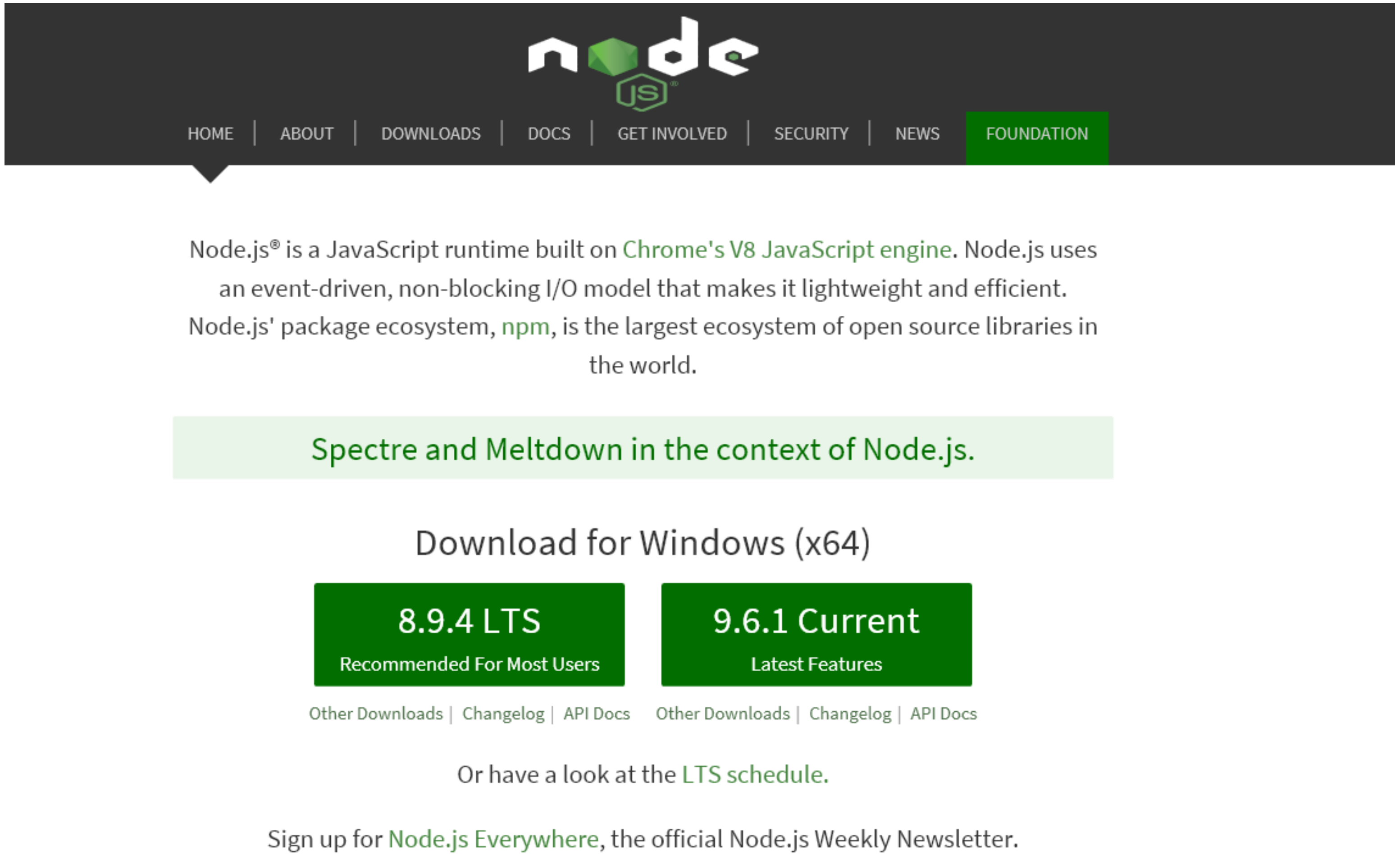


IF we remove the first double quote we get a JSON.parse(person2) error

```
const person2 = `
{
    firstName": "sally",
    "lastName" : "jones"
}
`; // NOTICE these are back ticks and not single quotes. Back ticks allow
   // us to put white space and content across several lines
console.log(person2); // print out actually what we have above between the braces

// lets create a Javascript object from JSON
var person3 = JSON.parse(person2)
```

```
undefined:3
    firstName": "sally",
    ^

SyntaxError: Unexpected token f in JSON at position 7
    at JSON.parse (<anonymous>)
```

The Tooling Nodejs

- We need Nodejs as all the tooling and support of packages are shipped and maintained via NPM (Node Package Manager)
- https://nodejs.org/en/
- 

HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | NEWS | FOUNDATION

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

Spectre and Meltdown in the context of Node.js.

Download for Windows (x64)

8.9.4 LTS
Recommended For Most Users

9.6.1 Current
Latest Features

Other Downloads | Changelog | API Docs   Other Downloads | Changelog | API Docs

Or have a look at the LTS schedule.

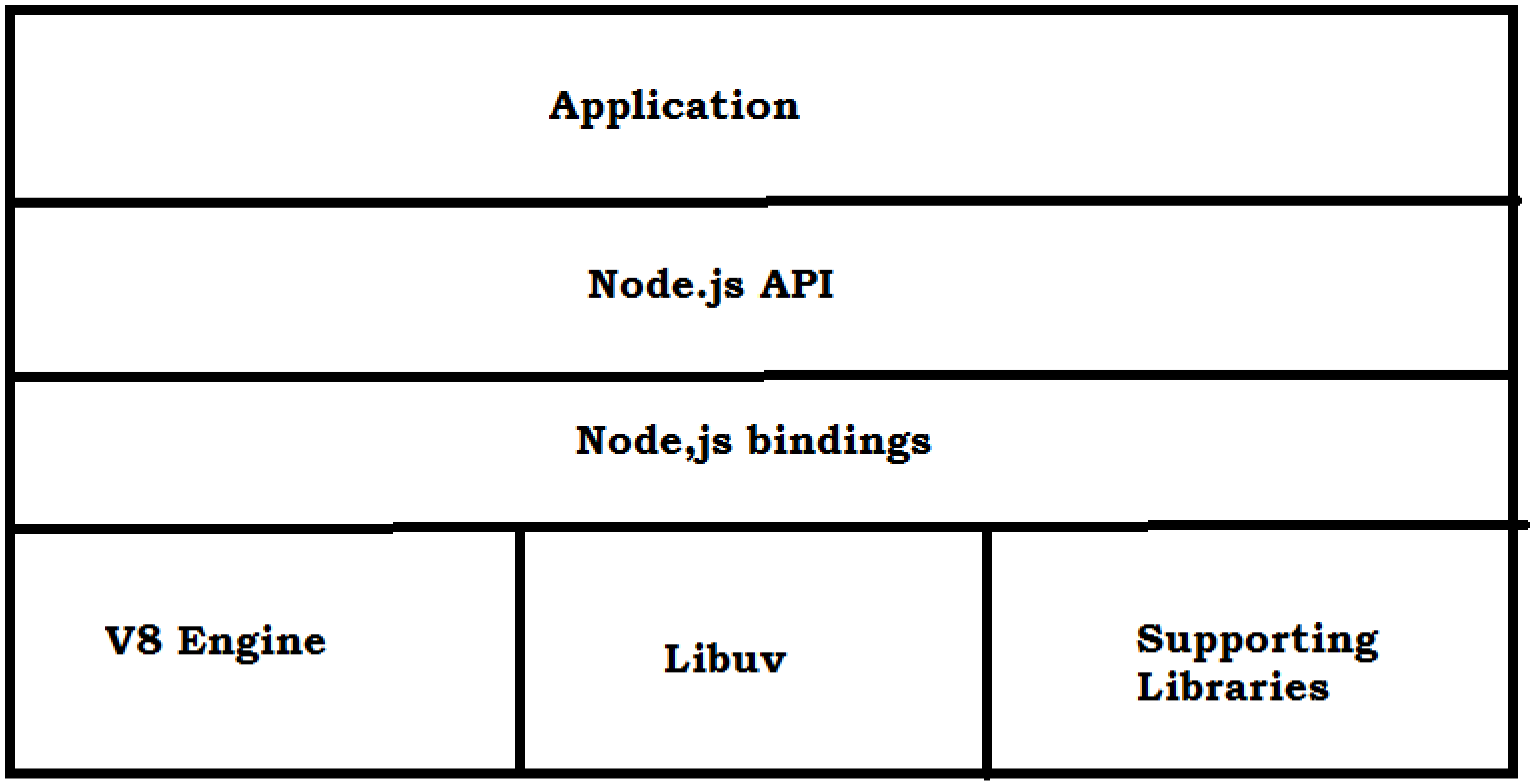
Sign up for Node.js Everywhere, the official Node.js Weekly Newsletter.

Blank Nodejs Project

- Downloaded and installed nodejs
- Create an app directory and cd into i.e. c:\projects\app\
- Then npm to initialize an package.json file with defaults
- npm init --yes
- --yes gives you a default answer

What is Nodejs

- Wikipedia states: “Node.js is a packaged compilation of Google’s V8 JavaScript engine, the libuv platform abstraction layer, and a core library, which is itself primarily written in JavaScript.” Beyond that, it’s worth noting that Ryan Dahl, the creator of Node.js, was aiming to create real-time websites with push capability, “inspired by applications like Gmail”. In Node.js, he gave developers a tool for working in the non-blocking, event-driven I/O paradigm.


Application
Node.js API
Node,js bindings
V8 Engine
Libuv
Supporting Libraries

What is Nodejs

- **License** – Node.js is released under MIT License
- Asynchronous and Event Driven – All APIs of Node.js library are asynchronous, that is, non-blocking.
- The Engine was built on Google Chrome's V8 JavaScript Engine, but you can use a different Engine like Microsoft’s Javascript Engine
- Single Threaded but Highly Scalable – Node.js uses a single threaded model with event looping

Nodejs Use Cases

- Serving Single Page Applications
- Data Intensive Real-time Applications (DIRT)
- Restful JSON APIs based Applications
- Data Streaming Applications
- I/O bound Applications
- General Web Server Behind Proxy
- Good Support by Cloud providers for Lambda function, MicroServices, Containers i.e. Docker.
- Support for Native WebSockets and extension frameworks like Socket.io
- Realtime Communications application like WebRTC, Chat, IRC etc.

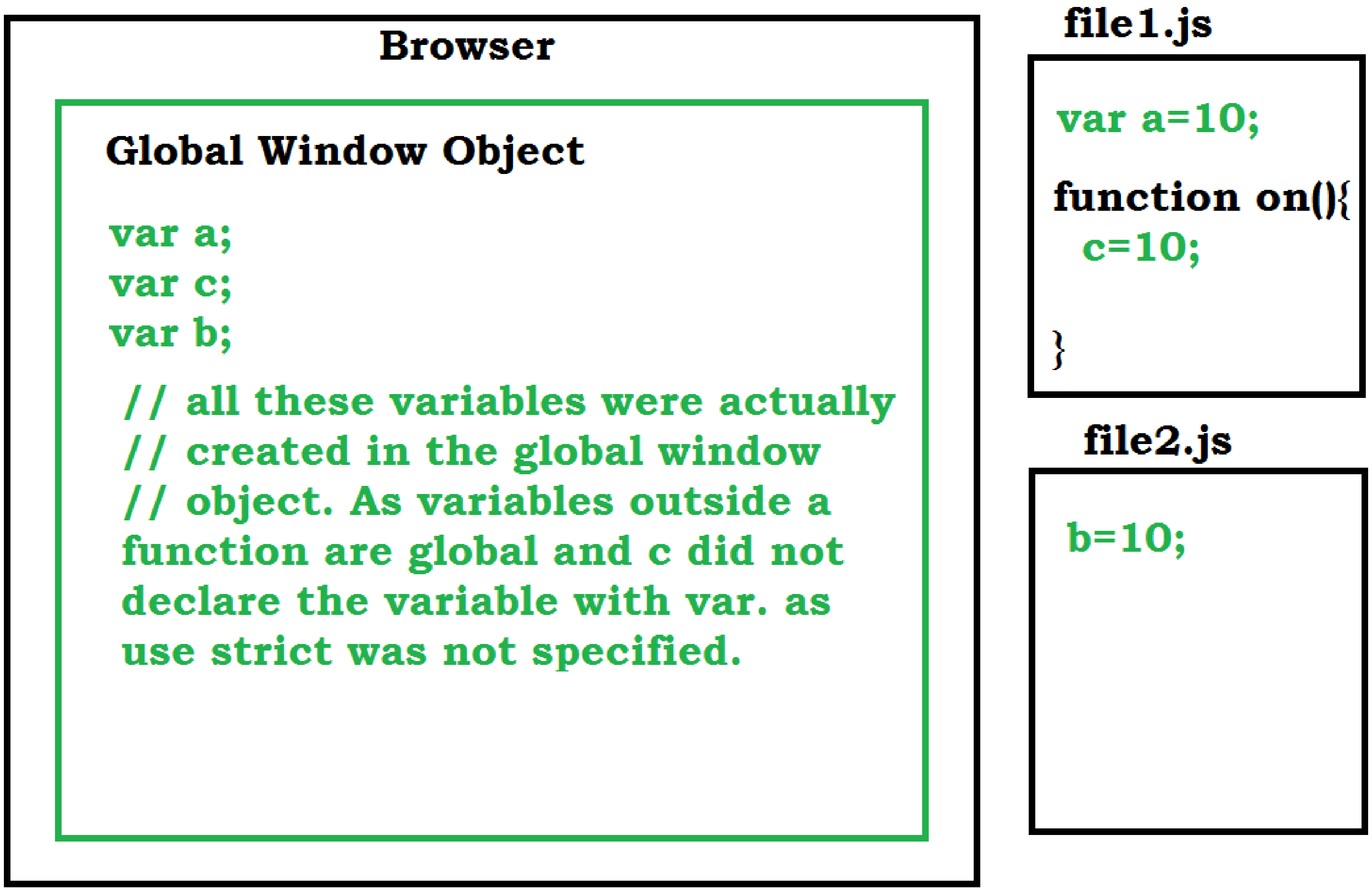
Nodejs vs Web Browsers

Nodejs

- Treats a file as module scope and not global scope
- with built-in Module System i.e - (require/exports) CommonJS (CJS) which Node.js has used historically and supports ES6 Import/exports keywords. Node will also add support for ESM modules
- Has explicit Global objects. i.e. Global, Process etc
- Has Great NPM Package Manager support and other packkage managers like yarn.
- Supports web workers and Clustering via fork processes NPM package

Web Browsers

- (ESM) EcmaScript modules Modules coming/added to new browsers in 2017/2018
- By default and variables are placed in the global space i.e. called the window
- Support web workers

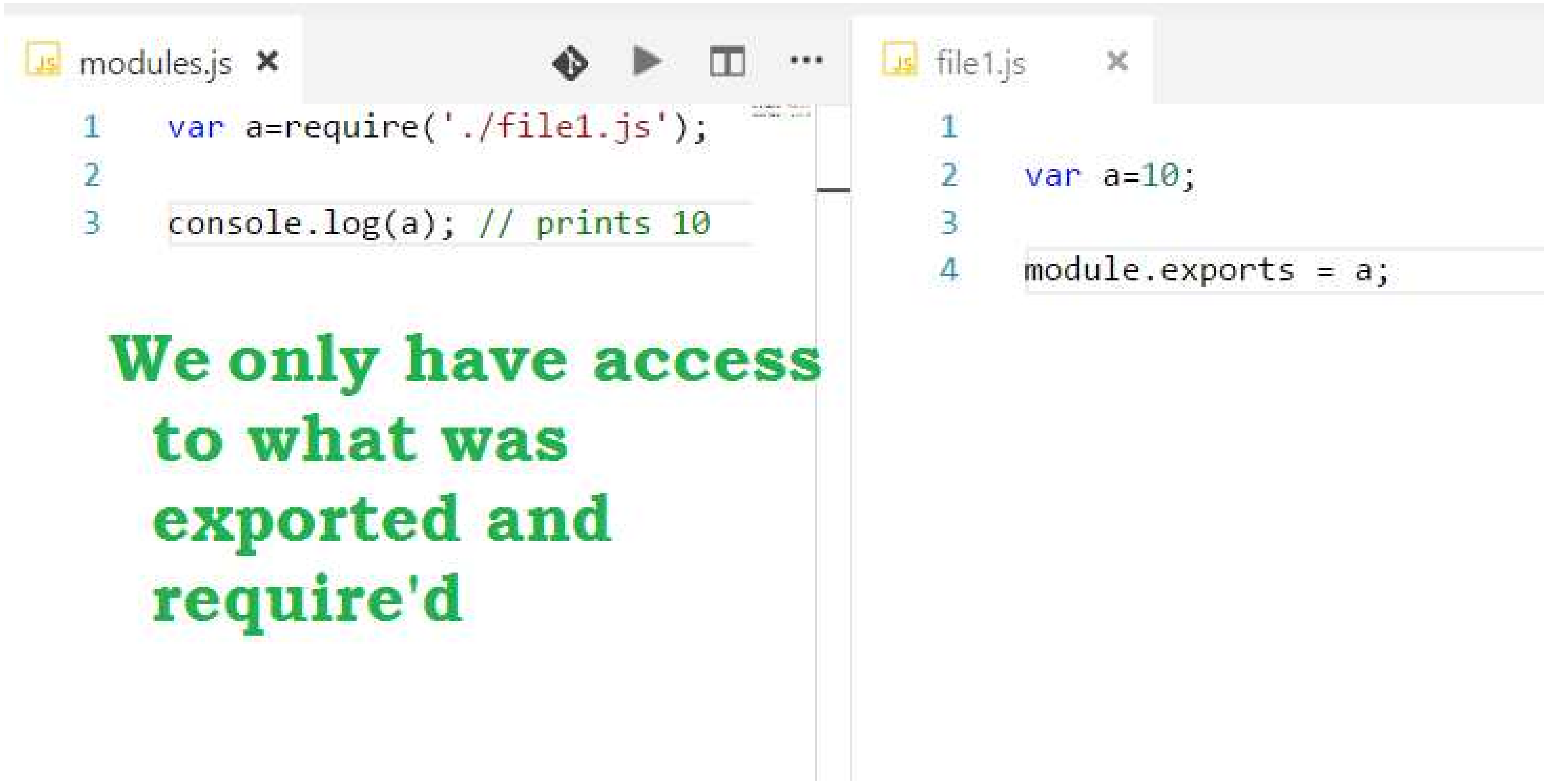


```
var a;
var c;
var b;
 // all these variables were actually
 // created in the global window
 // object. As variables outside a
 function are global and c did not
 declare the variable with var. as
 use strict was not specified.
```

```
var a=10;

function on(){
  c=10;

}
```

```
b=10;
```

Node’s default File Module Wrapping

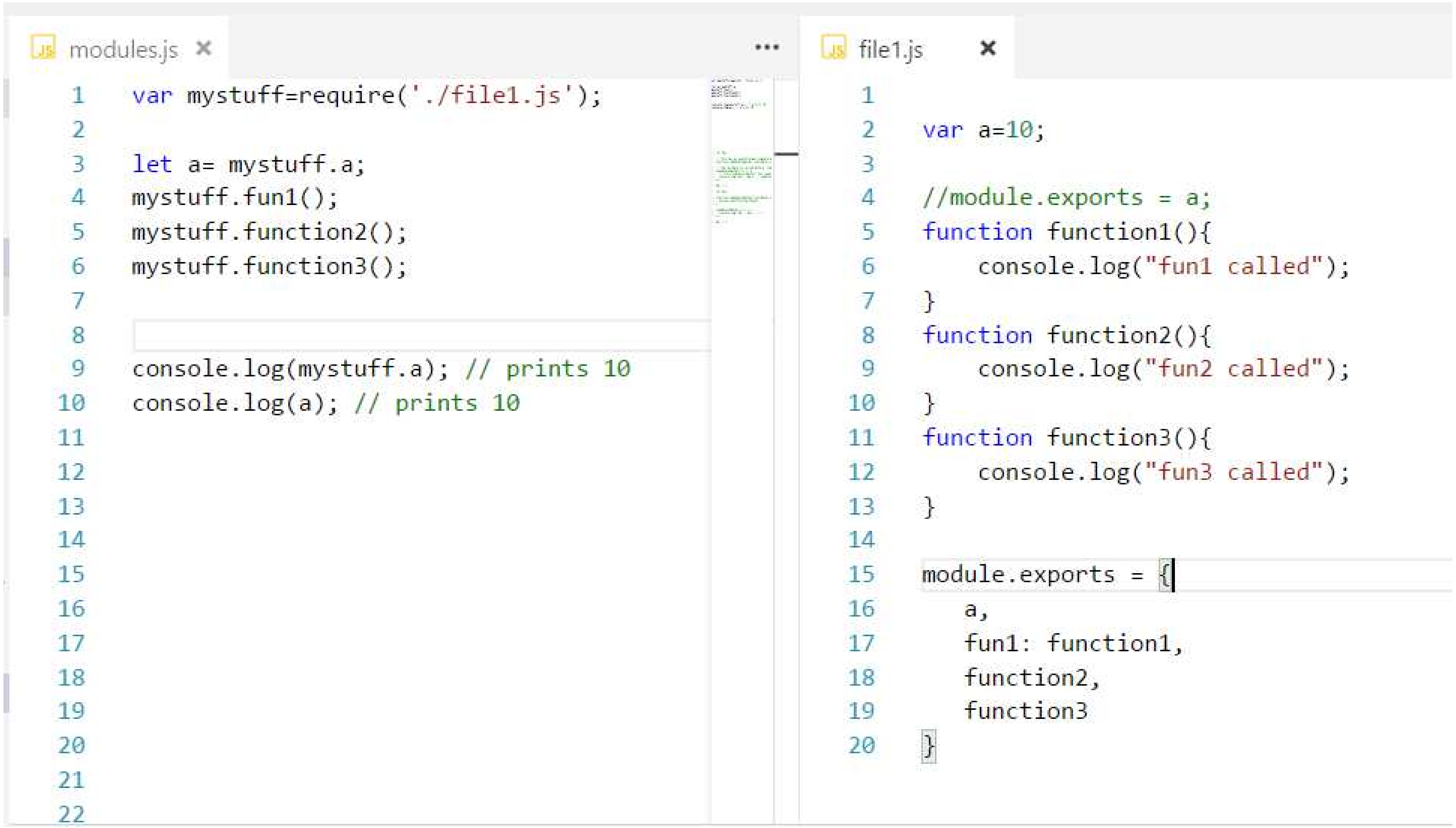


```
var a=require('./file1.js');

console.log(a); // prints 10
```

```

var a=10;

module.exports = a;
```

We only have access to what was exported and require'd

How to export several variables/functions/classes/objects



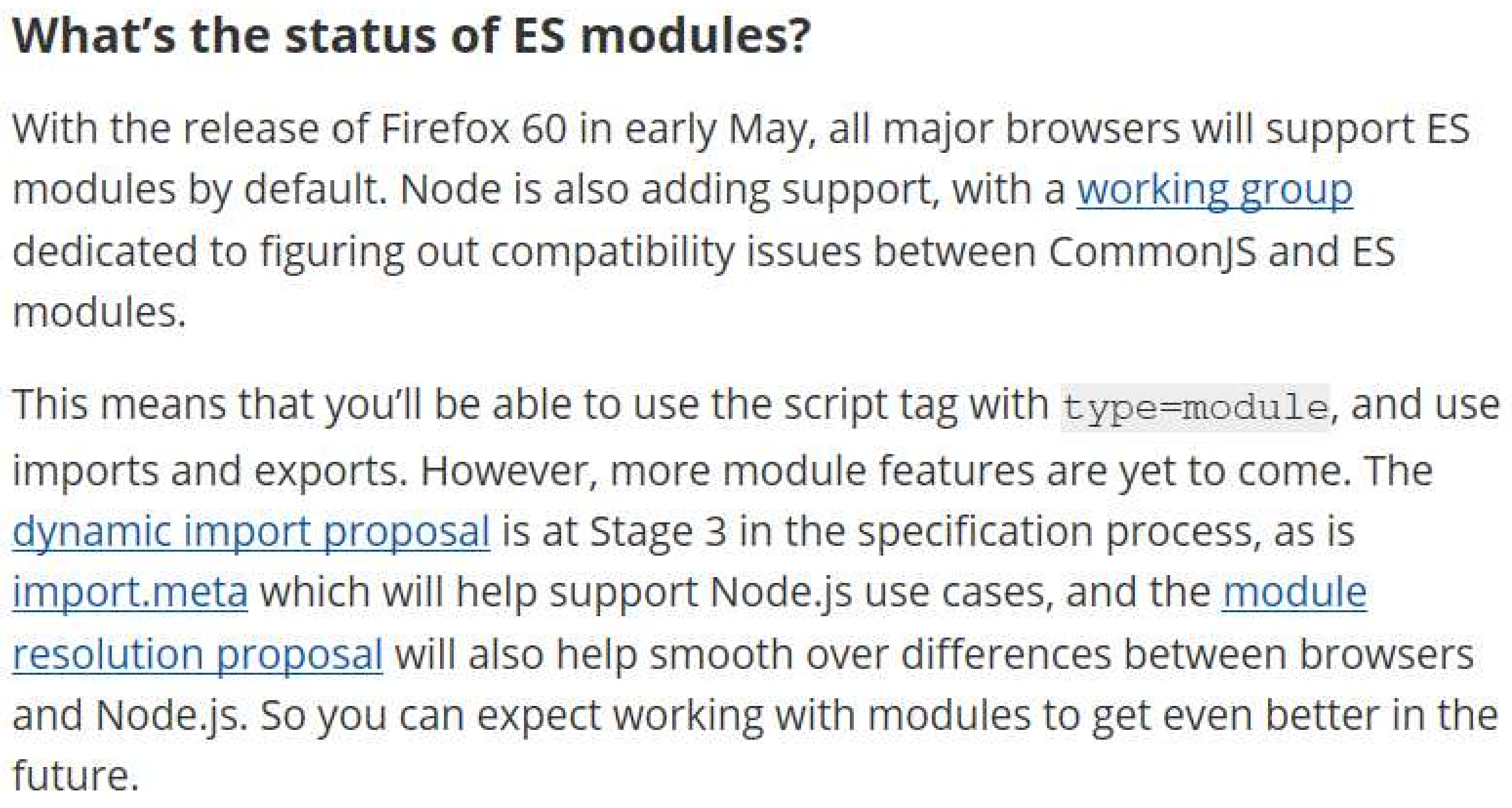
```
1   var mystuff=require('./file1.js');
2
3   let a= mystuff.a;
4   mystuff.fun1();
5   mystuff.function2();
6   mystuff.function3();
7
8
9   console.log(mystuff.a); // prints 10
10  console.log(a); // prints 10
11
12
13
14
15
16
17
18
19
20
21
22
```

```
1
2   var a=10;
3
4   //module.exports = a;
5   function function1(){
6       console.log("fun1 called");
7   }
8   function function2(){
9       console.log("fun2 called");
10  }
11  function function3(){
12      console.log("fun3 called");
13  }
14
15  module.exports = {
16     a,
17     fun1: function1,
18     function2,
19     function3
20  }
```

Line 18 and 19 Object ES6 shortcuts : function2 automatically refers to the function function2

ES modules By Lin Clark

- https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/



What's the status of ES modules?

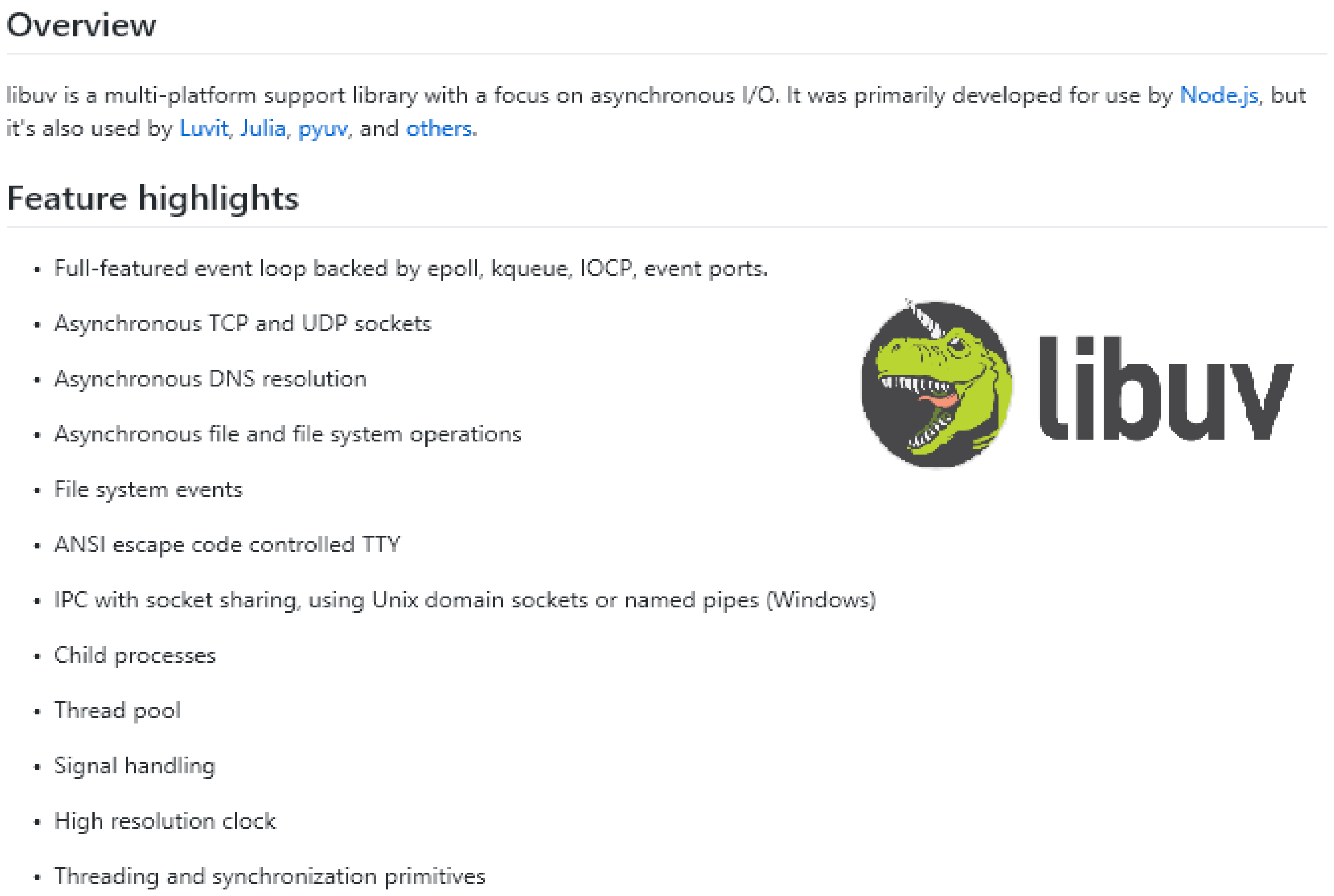
With the release of Firefox 60 in early May, all major browsers will support ES modules by default. Node is also adding support, with a working group dedicated to figuring out compatibility issues between CommonJS and ES modules.

This means that you'll be able to use the script tag with `type=module`, and use imports and exports. However, more module features are yet to come. The dynamic import proposal is at Stage 3 in the specification process, as is import.meta which will help support Node.js use cases, and the module resolution proposal will also help smooth over differences between browsers and Node.js. So you can expect working with modules to get even better in the future.

The Event Loop

- So is NodeJS really single threaded?
- How are events handled like Timers, IO, Sockets
- You can set the Nodejs Thread Pool with the Env variable . This is for libuv
- UV_THREADPOOL_SIZE=64 node
- Or in your node code with
- process.env.UV_THREADPOOL_SIZE=64
- The thread pool is used mainly for IO i.e. modules like fs etc. opening,reading,writing
- Cluster processes get their own Event Loop

Libuv provides nodejs's Event Loop - https://github.com/libuv/libuv

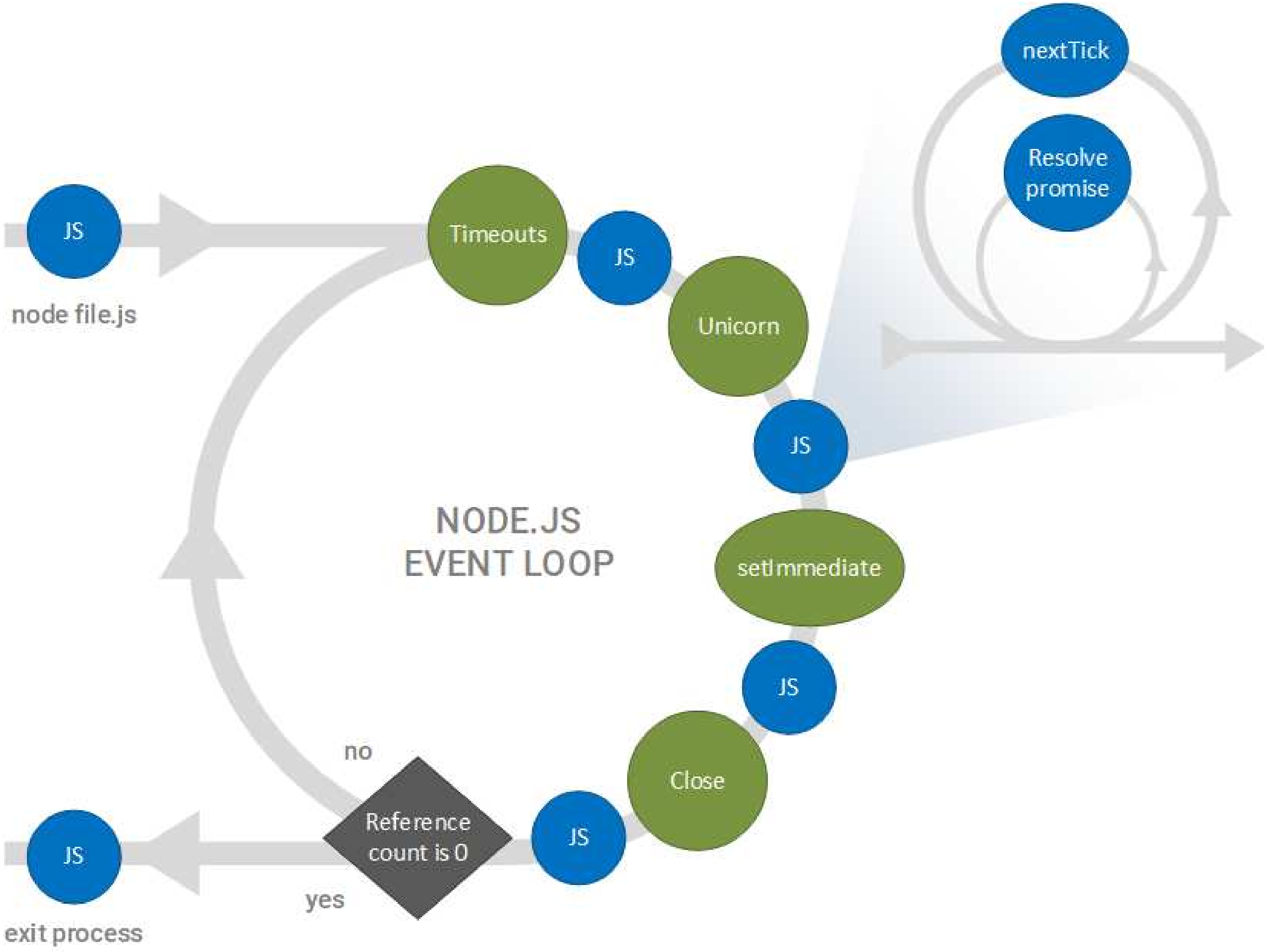


Overview

libuv is a multi-platform support library with a focus on asynchronous I/O. It was primarily developed for use by Node.js, but it's also used by Luvit, Julia, pyuv, and others.

Feature highlights

- Full-featured event loop backed by epoll, kqueue, IOCP, event ports.
- Asynchronous TCP and UDP sockets
- Asynchronous DNS resolution
- Asynchronous file and file system operations
- File system events
- ANSI escape code controlled TTY
- IPC with socket sharing, using Unix domain sockets or named pipes (Windows)
- Child processes
- Thread pool
- Signal handling
- High resolution clock
- Threading and synchronization primitives



https://dzone.com/articles/introduction-to-nodejs-3

Lets talk about Timers in the Event Loop

- Let’s mention some Timer/Tick callback functions
- function myFunc(name){ console.log(name); }

- // let and const was introduced in ES6
- const cancelSTO = setTimeout(myFunc, 1000, ‘george');
- clearTimeout(cancelSTO);

- let cancelSI = setImmediate(myFunc , ‘george');
- clearImmediate(cancelSI);

- const intervalObj = setInterval(myFunc, 500, ‘george’);
- process.nextTick(myFunc, ‘george’);

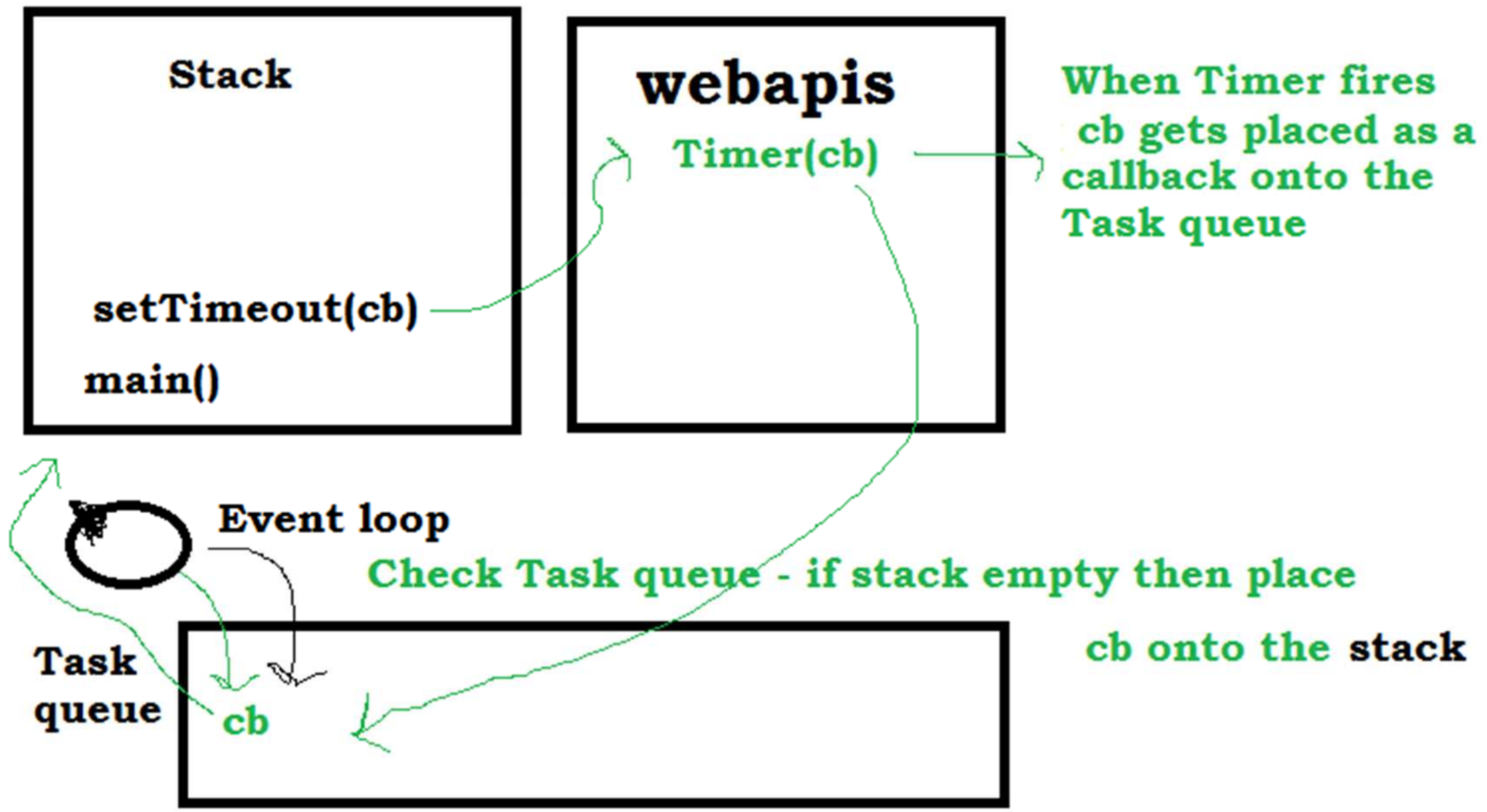
The following diagram shows a simplified overview of the event loop's order of operations.

```
   ┌───────────────────────┐
┌─>│        timers         │
│  └──────────┬────────────┘
│  ┌──────────┴────────────┐
│  │     I/O callbacks     │
│  └──────────┬────────────┘
│  ┌──────────┴────────────┐
│  │     idle, prepare     │
│  └──────────┬────────────┘      ┌───────────────┐
│  ┌──────────┴────────────┐      │   incoming:   │
│  │         poll          │<─────┤  connections, │
│  └──────────┬────────────┘      │   data, etc.  │
│  ┌──────────┴────────────┐      └───────────────┘
│  │        check          │
│  └──────────┬────────────┘
│  ┌──────────┴────────────┐
└──┤    close callbacks    │
   └───────────────────────┘
```

note: each box will be referred to as a "phase" of the event loop.

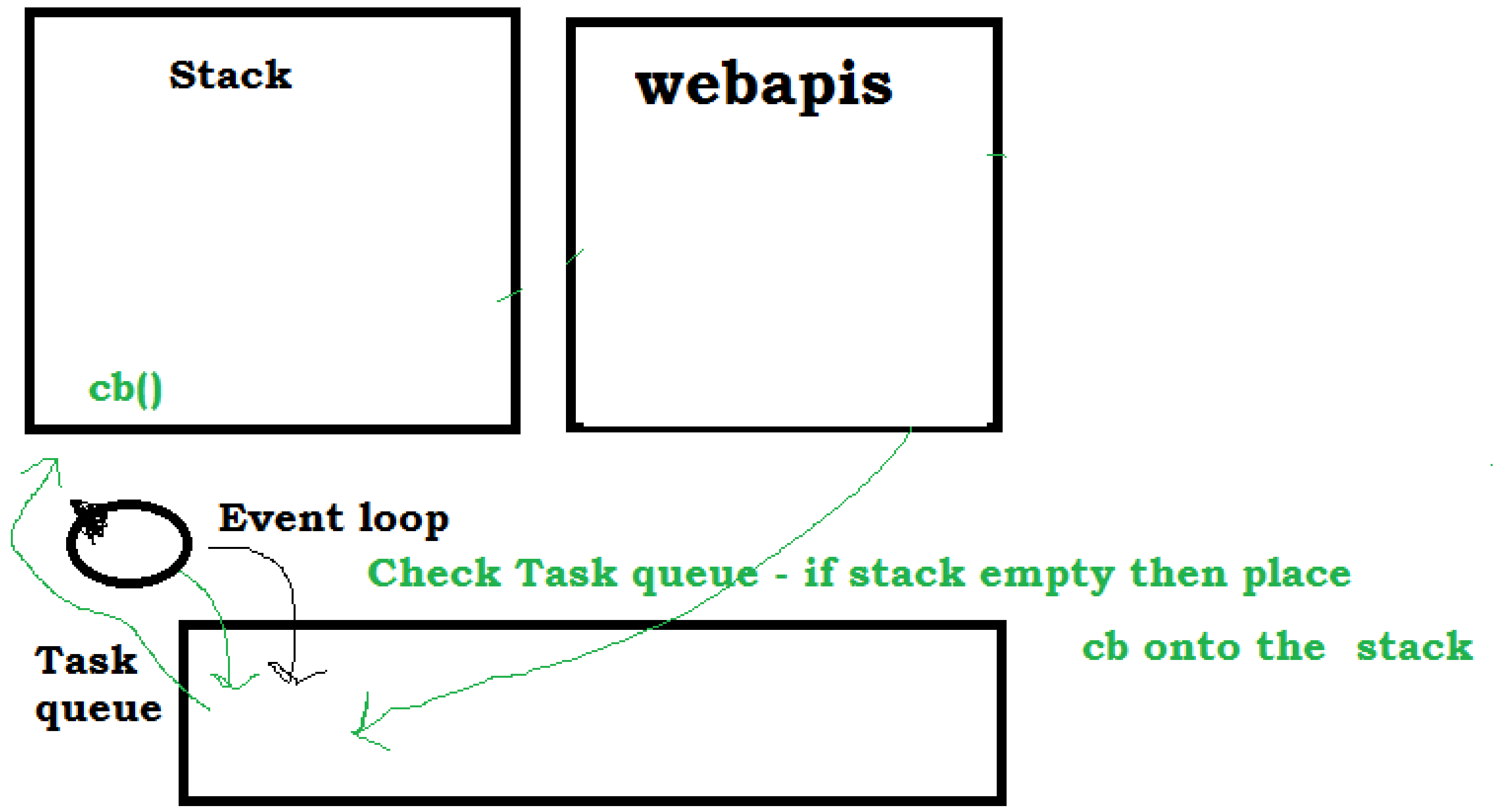
How people picture Browser event loops/ unrelated to nodejs/libuv

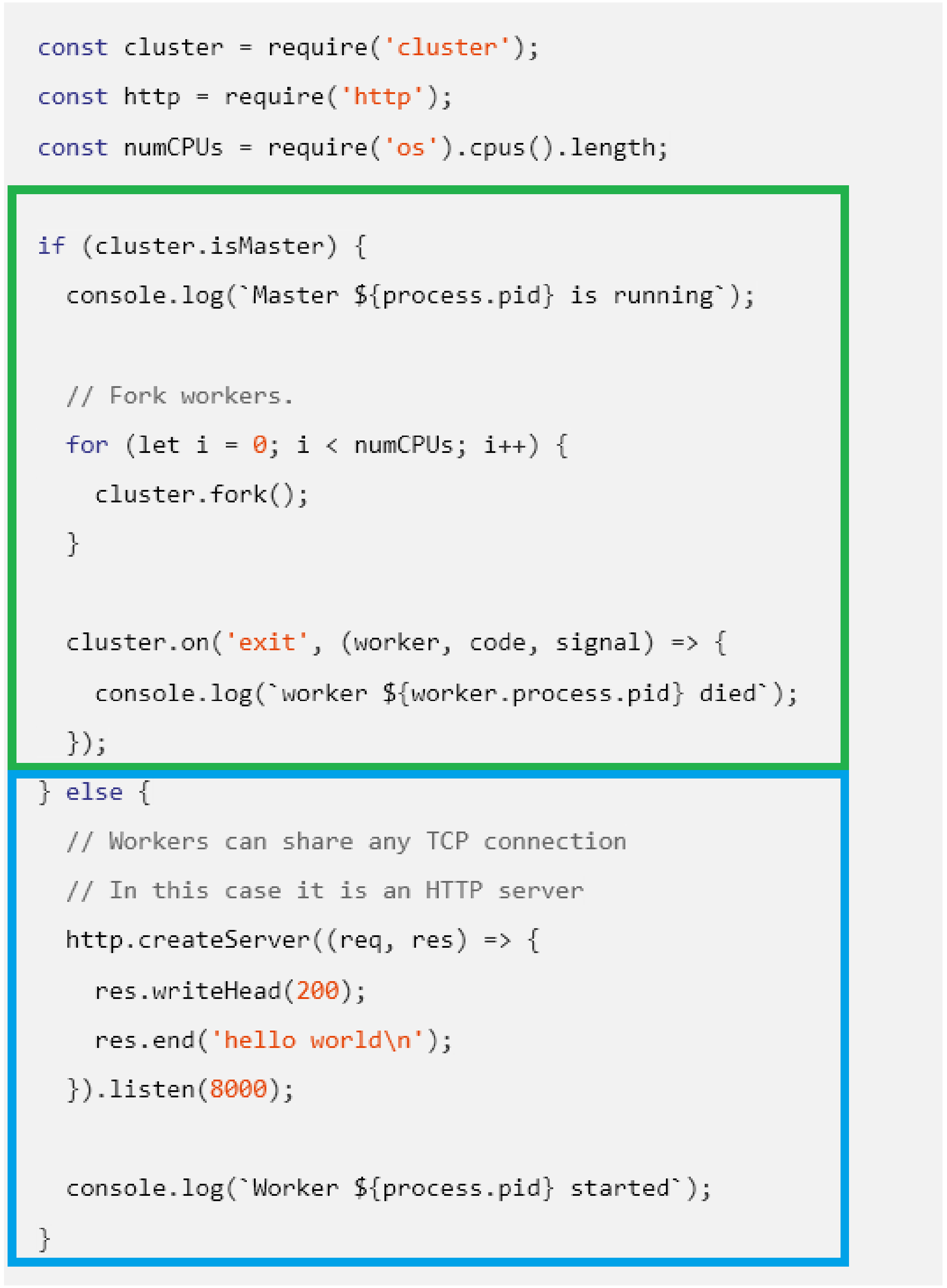
People refer to how browsers work at a top level when trying to describe how it's Event loop works. But of course don't confuse nodejs i.e. libuv with a browser.


Stack
setTimeout(cb)
main()
webapis
Timer(cb)
When Timer fires cb gets placed as a callback onto the Task queue
Event loop
Check Task queue - if stack empty then place cb onto the stack
Task queue
cb

How people picture Browser event loops/ unrelated to nodejs/libuv

People refer to how browsers work at a top level when trying to describe how it's Event loop works. But of course don't confuse nodejs i.e. libuv with a browser.





```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);

  console.log(`Worker ${process.pid} started`);
}
```

<u>Example of the cluster Module</u>

The cluster module supports two methods of distributing incoming connections.

The first one (and the default one on all platforms except Windows), is the round-robin approach, where the master process listens on a port, accepts new connections and distributes them across the workers in a round-robin fashion, with some built-in smarts to avoid overloading a worker process.

The second approach is where the master process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly.

```
var app = require('http').createServer(handler)
var io = require('socket.io')(app);
var fs = require('fs');

app.listen(80);
```

notice app= is where we create an instance of our http server and pass in the function we will use to handle requests

```
function handler (req, res) {
  fs.readFile(__dirname + '/index.html',
  function (err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading index.html');
    }

    res.writeHead(200);
    res.end(data);
  });
}
```

basic web server handler function

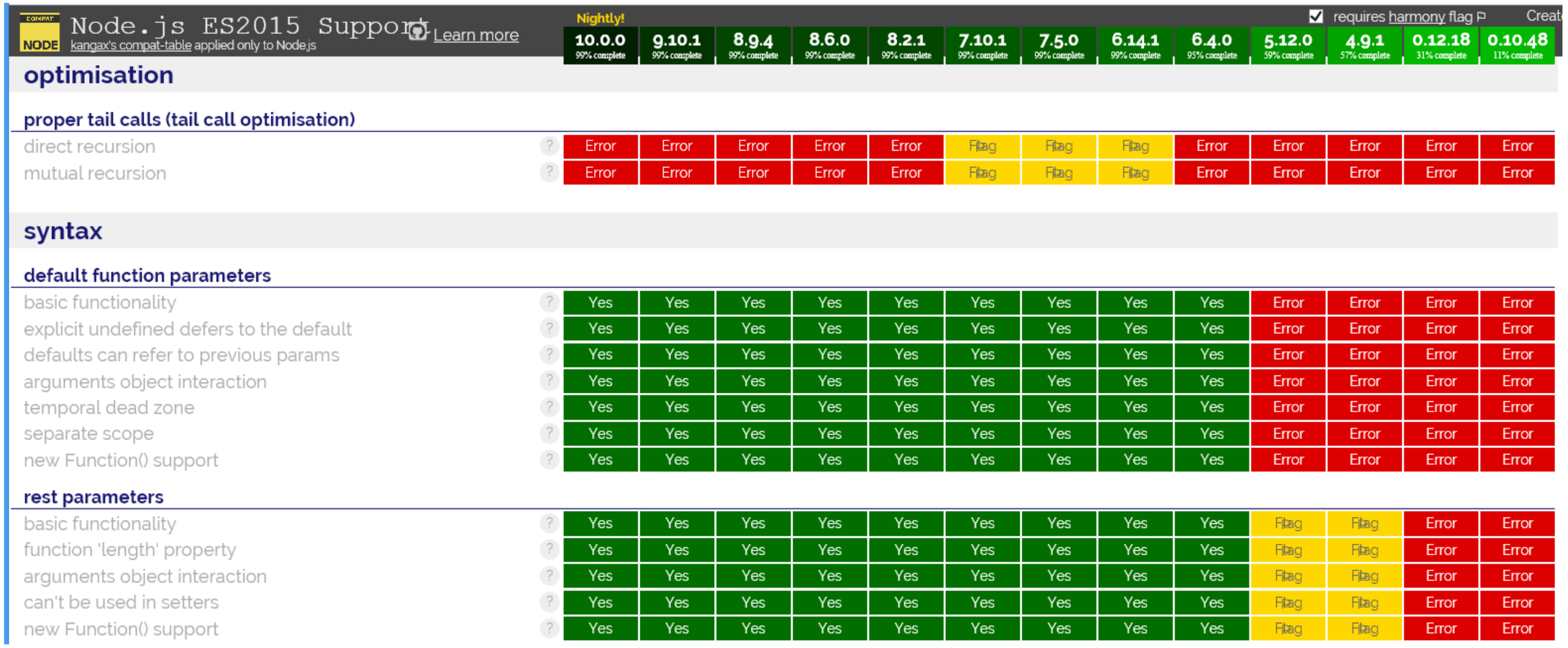
```
io.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });
  socket.on('my other event', function (data) {
    console.log(data);
  });
});
```

setup our socket.io events

Client (index.html)

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io('http://localhost');
  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'data' });
  });
</script>
```

ES2015/ES2016/ES2017
ES6/ES7/ES8 Javascript support
https://node.green/



Node.js ES2015 Support Learn more
kangax's compat-table applied only to Node.js

Nightly!

requires harmony flag

Creat

	10.0.0 99% complete	9.10.1 99% complete	8.9.4 99% complete	8.6.0 99% complete	8.2.1 99% complete	7.10.1 99% complete	7.5.0 99% complete	6.14.1 99% complete	6.4.0 95% complete	5.12.0 59% complete	4.9.1 57% complete	0.12.18 31% complete	0.10.48 11% complete
optimisation													
proper tail calls (tail call optimisation)													
direct recursion	Error	Error	Error	Error	Error	Flag	Flag	Flag	Error	Error	Error	Error	Error
mutual recursion	Error	Error	Error	Error	Error	Flag	Flag	Flag	Error	Error	Error	Error	Error
syntax													
default function parameters													
basic functionality	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Error	Error	Error	Error
explicit undefined defers to the default	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Error	Error	Error	Error
defaults can refer to previous params	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Error	Error	Error	Error
arguments object interaction	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Error	Error	Error	Error
temporal dead zone	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Error	Error	Error	Error
separate scope	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Error	Error	Error	Error
new Function() support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Error	Error	Error	Error
rest parameters													
basic functionality	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Flag	Error	Error
function 'length' property	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Flag	Error	Error
arguments object interaction	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Flag	Error	Error
can't be used in setters	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Flag	Error	Error
new Function() support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Flag	Error	Error

We will only look at ES6/ES2015

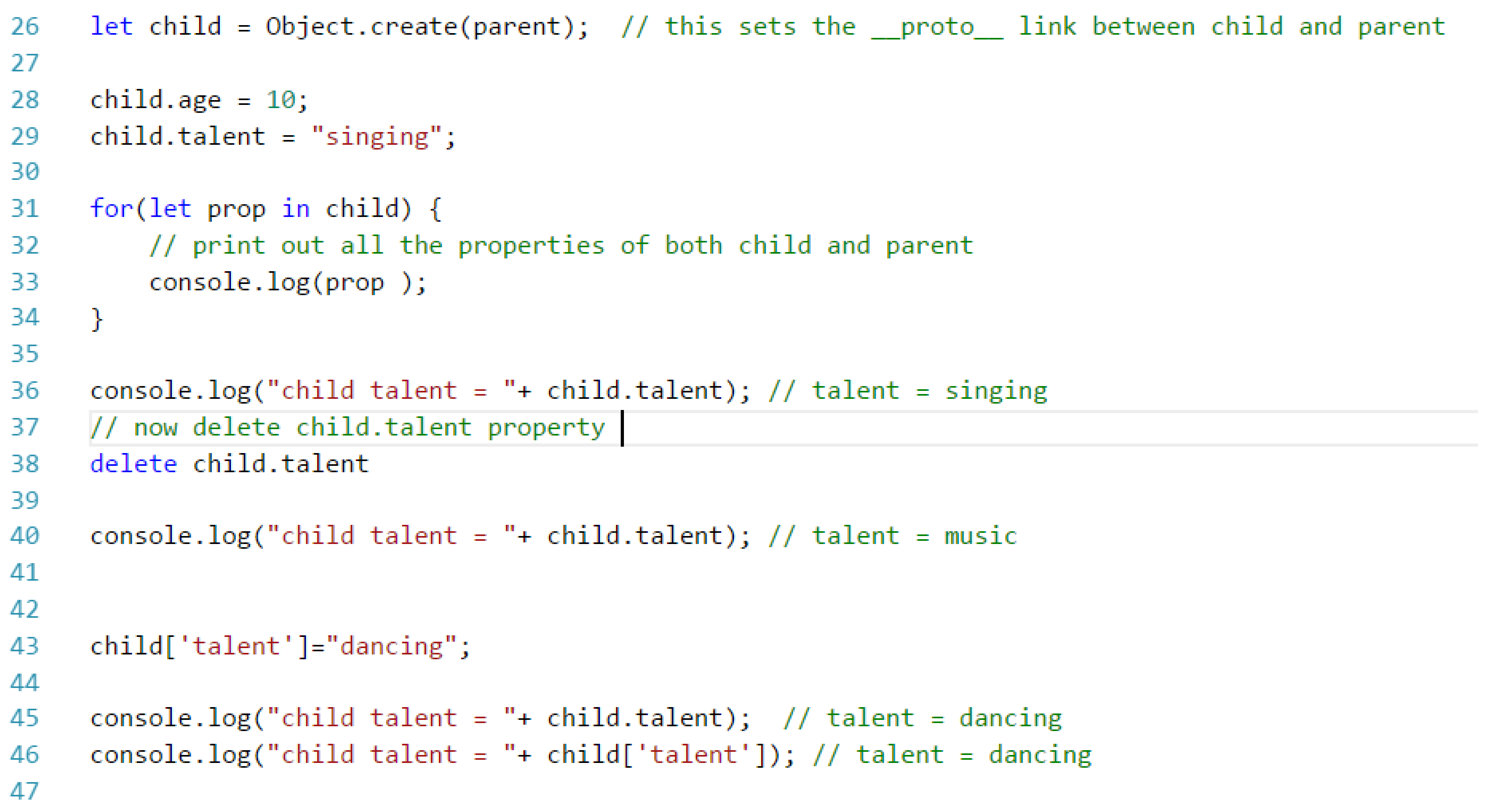
The Basics

- So Javascript basics first
- Lets go through 4 ways to create an Javascript Object.
- Why so many different ways
- Look at the ES6 class and how the class is syntactic sugar with standard javascript underneath.

The Object Literal

```javascript

let parent = {

    firstName : "sally",
    lastName  : "smith",
    age : 32,
    "secret code" : "bananas",
    address   : {
        street : "123 o'connell st",
        city   : "Limerick"
    },
    talent : "music",
    fullName (){
       return this.firstName + this.lastName;
    }
}

console.log("parent age =" + parent.age);
console.log("parent secret code =" + parent['secret code']);
console.log("address.street  =" + parent.address.street);
console.log("address.street  =" + parent['address']['street']);
console.log("address.street  =" + parent['address'].street);

```

Now lets use CreateObject to show Javascript prototype chaining and delete

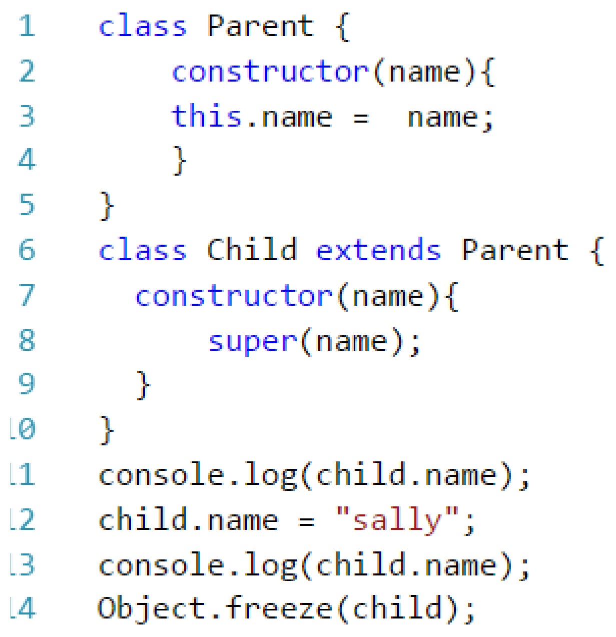


```
let child = Object.create(parent);  // this sets the __proto__ link between child and parent

child.age = 10;
child.talent = "singing";

for(let prop in child) {
    // print out all the properties of both child and parent
    console.log(prop );
}

console.log("child talent = "+ child.talent); // talent = singing
// now delete child.talent property
delete child.talent

console.log("child talent = "+ child.talent); // talent = music


child['talent']="dancing";

console.log("child talent = "+ child.talent);  // talent = dancing
console.log("child talent = "+ child['talent']); // talent = dancing
```

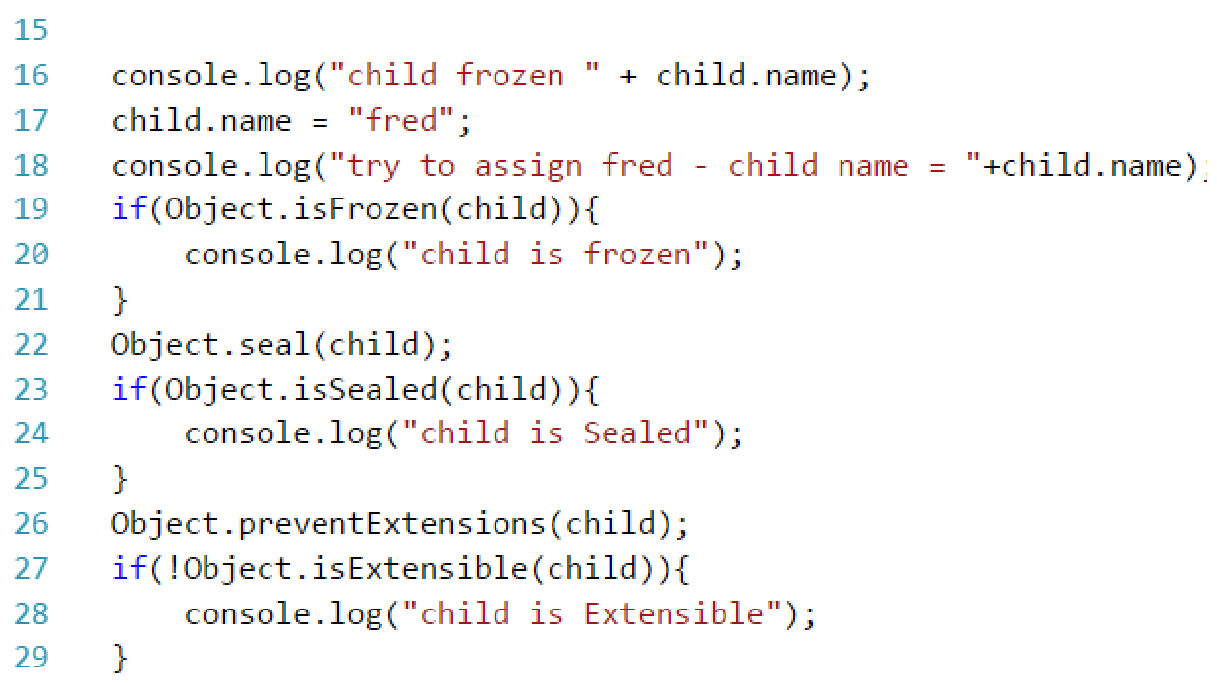
Object.defineProperty in ES5/ES6

```javascript
Object.defineProperty(child, 'height', {
   // Both data and accessor descriptors are objects
   // they share the following keys
   //  configurable : true if and only if the type of this property descriptor may be changed and
   //  if the property may be deleted from the corresponding object.
   //   Defaults to false.
   configurable : true,

   // enumerable: if and only if this property shows up during enumeration of the properties on the
   //  corresponding object.
   // Defaults to false.
   enumerable : true,

   // a data descriptor also has the following optonal keys

   // The value associated with the property. Can be any valid JavaScript value (number, object, function, etc).
   // Defaults to undefined.
   value : 0,

   // writabletrue if and only if the value associated with the property may be changed with an assignment operator.
   // Defaults to false.
   writable:true,

   // an accessor descriptor has the following optional keys
   // get
   // A function which serves as a getter for the property, or
   // undefined if there is no getter.
   // set
   // A function which serves as a setter for the property, or undefined
   // if there is no setter.
});
```

Object.freeze(), Object.seal(), Object.preventExtensions()
Object.isFrozen(), Object.isSealed(), Object.isExtensible()



```
class Parent {
    constructor(name){
    this.name =  name;
    }
}
class Child extends Parent {
  constructor(name){
      super(name);
  }
}
console.log(child.name);
child.name = "sally";
console.log(child.name);
Object.freeze(child);
```

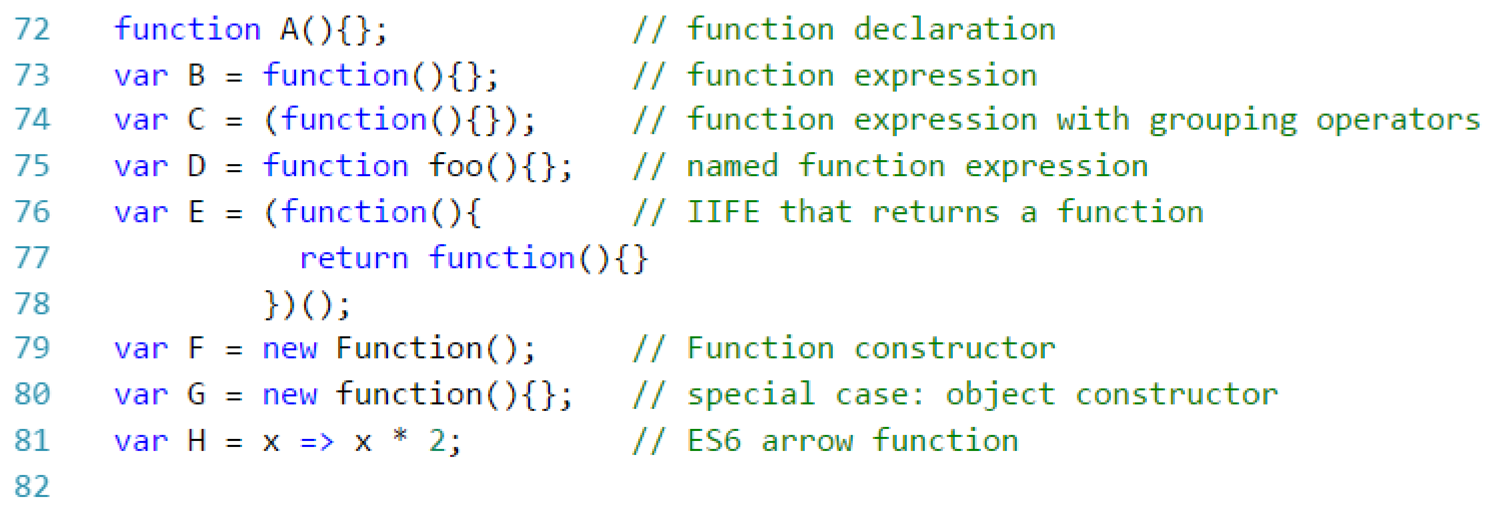


```

console.log("child frozen " + child.name);
child.name = "fred";
console.log("try to assign fred - child name = "+child.name);
if(Object.isFrozen(child)){
    console.log("child is frozen");
}
Object.seal(child);
if(Object.isSealed(child)){
    console.log("child is Sealed");
}
Object.preventExtensions(child);
if(!Object.isExtensible(child)){
    console.log("child is Extensible");
}
```

The Function Object

- People are confused by the many way to create Javascript objects.
- Functions serve 3 purposes in Javascript
- 1. As a unit of computation to accept an optional value, compute and return an optional value
- 2. Define a scope
- 3. Act as constructor of an object
- Constructor functions and classes both create a prototype chain.
- We have yet to discuss classes. But first let look at functions

This is just for fun = many functions

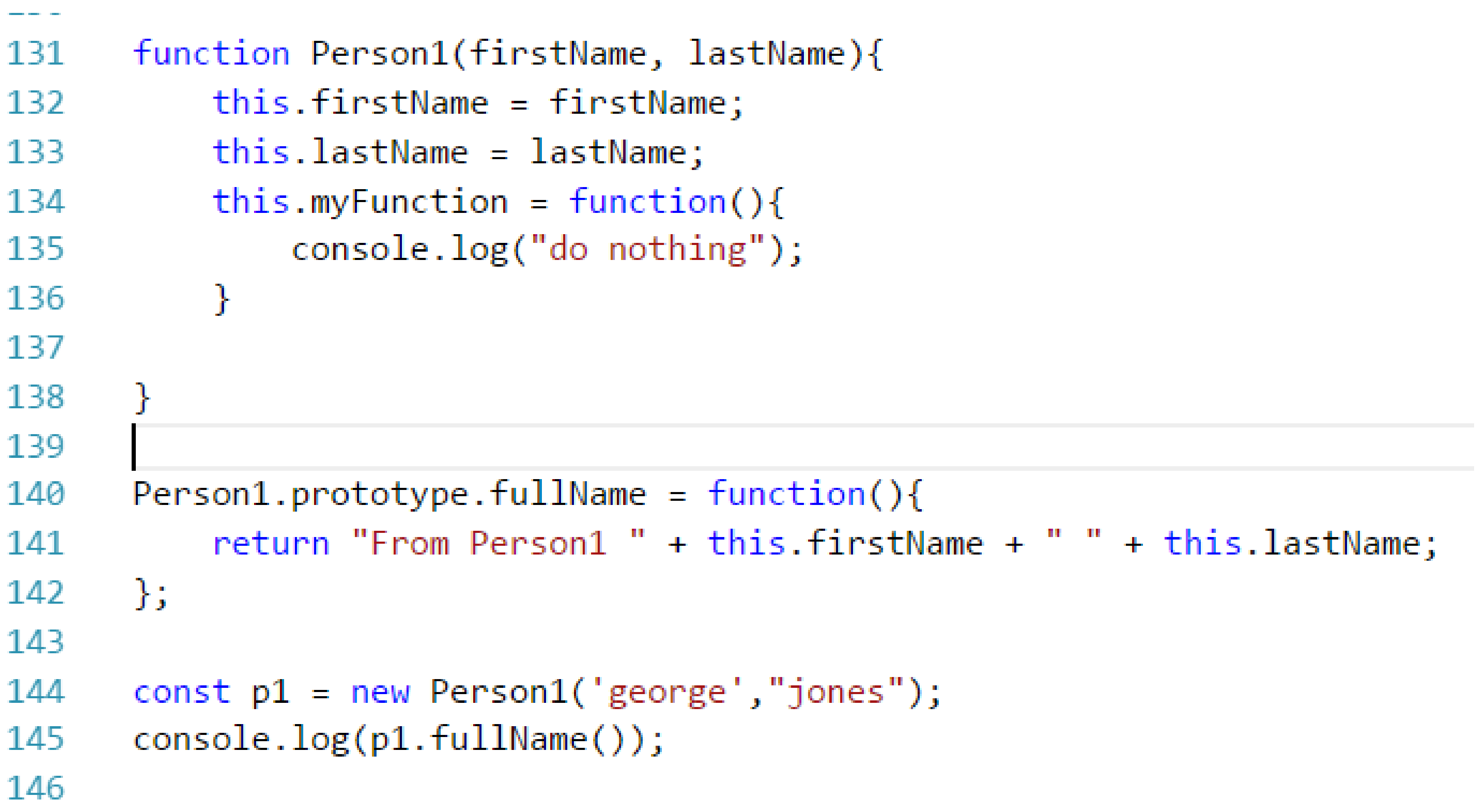


```
function A(){};             // function declaration
var B = function(){};       // function expression
var C = (function(){});     // function expression with grouping operators
var D = function foo(){};   // named function expression
var E = (function(){        // IIFE that returns a function
          return function(){}
        })();
var F = new Function();     // Function constructor
var G = new function(){};   // special case: object constructor
var H = x => x * 2;         // ES6 arrow function
```

When I wanted to compile the many ways you can create functions. I noticed an amusing blog post by David Calhoun which I coped the list from.

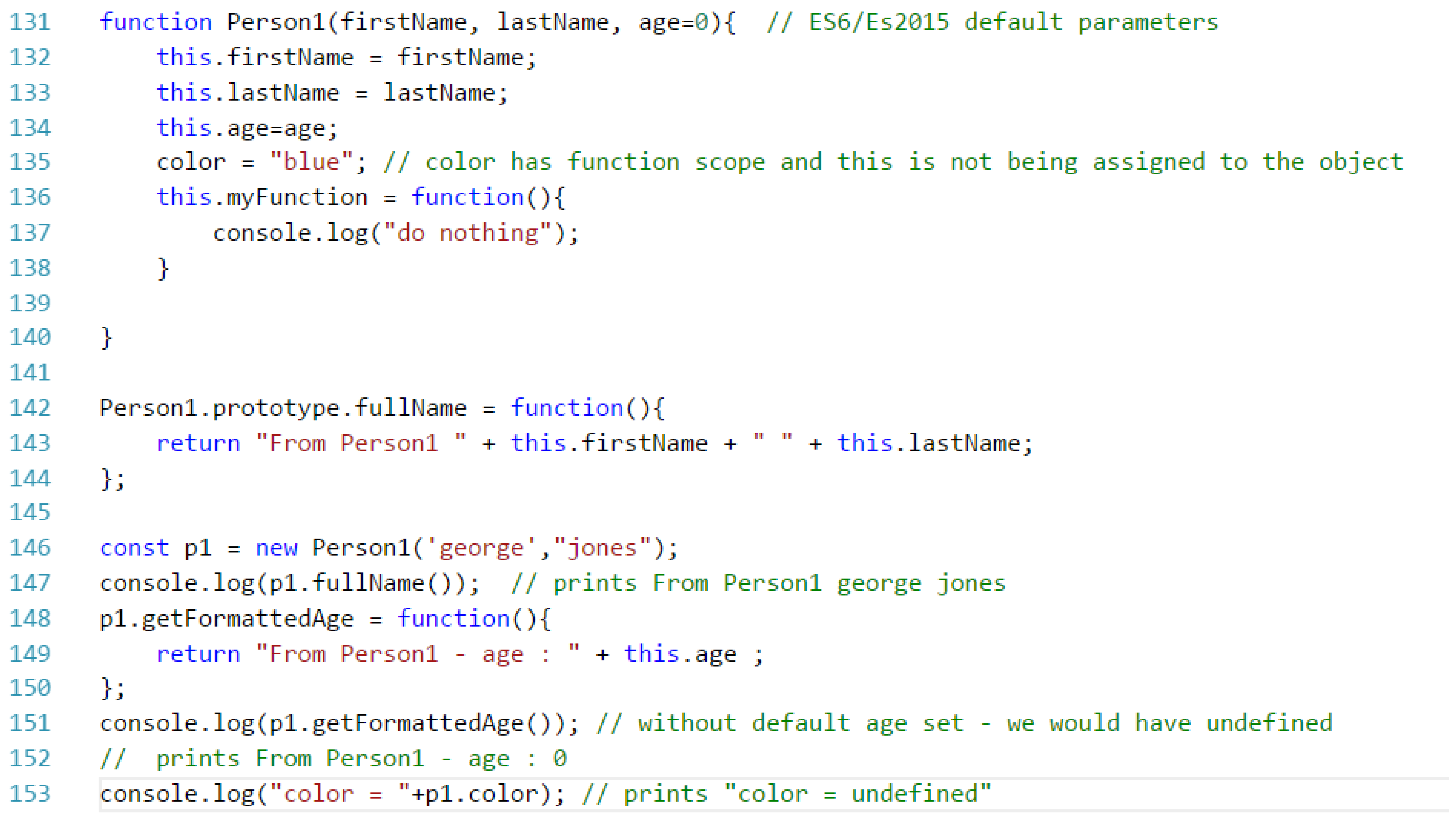
https://www.davidbcalhoun.com/2011/different-ways-of-defining-functions-in-javascript-this-is-madness/

Simple function constructor example



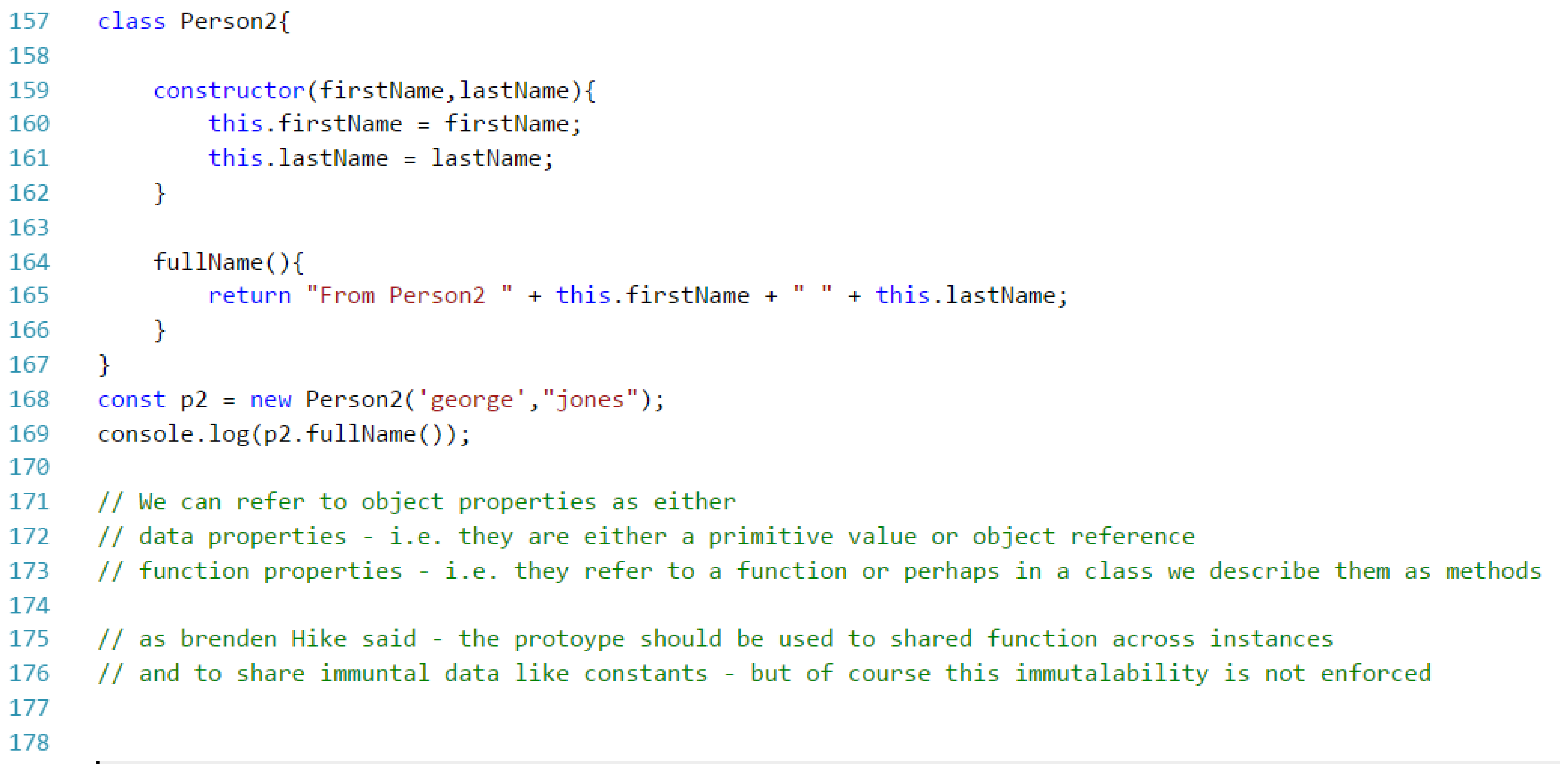
```
function Person1(firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
    this.myFunction = function(){
        console.log("do nothing");
    }

}

Person1.prototype.fullName = function(){
    return "From Person1 " + this.firstName + " " + this.lastName;
};

const p1 = new Person1('george',"jones");
console.log(p1.fullName());

```

Setting prototype and Default Param values



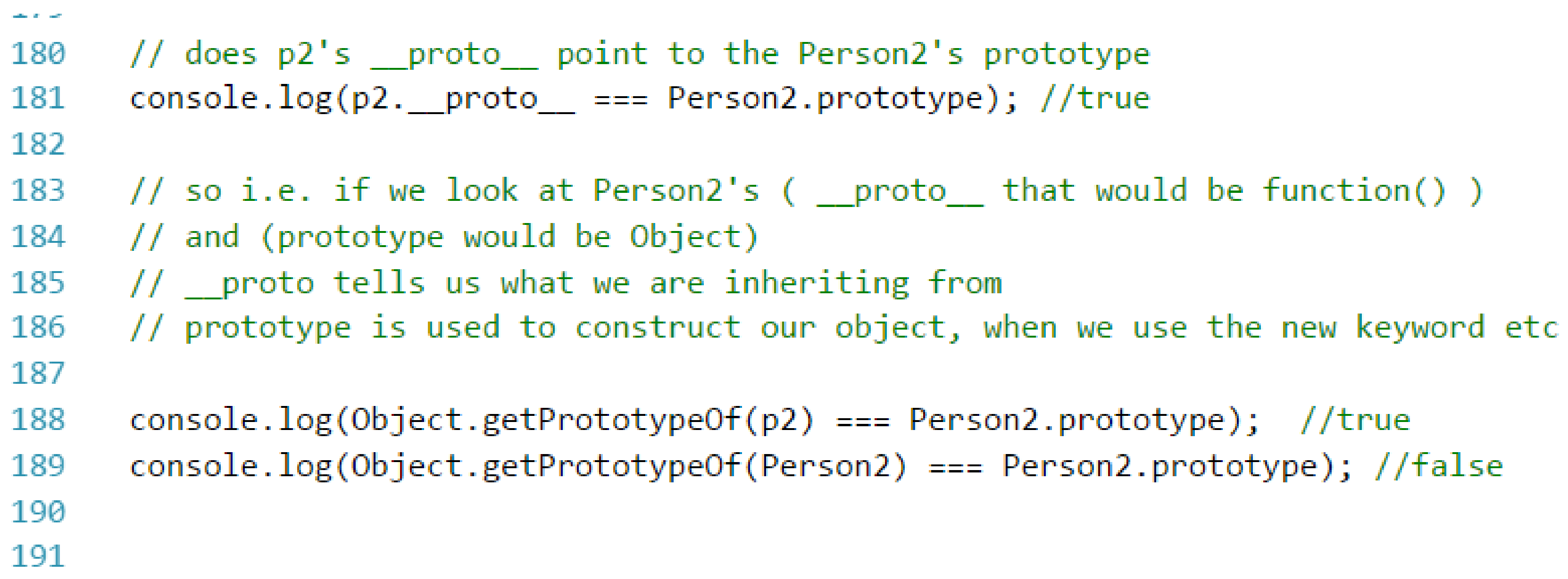
```javascript
function Person1(firstName, lastName, age=0){  // ES6/Es2015 default parameters
    this.firstName = firstName;
    this.lastName = lastName;
    this.age=age;
    color = "blue"; // color has function scope and this is not being assigned to the object
    this.myFunction = function(){
        console.log("do nothing");
    }

}

Person1.prototype.fullName = function(){
    return "From Person1 " + this.firstName + " " + this.lastName;
};

const p1 = new Person1('george',"jones");
console.log(p1.fullName());  // prints From Person1 george jones
p1.getFormattedAge = function(){
    return "From Person1 - age : " + this.age ;
};
console.log(p1.getFormattedAge()); // without default age set - we would have undefined
//  prints From Person1 - age : 0
console.log("color = "+p1.color); // prints "color = undefined"
```

ES6/ES2015 Classes



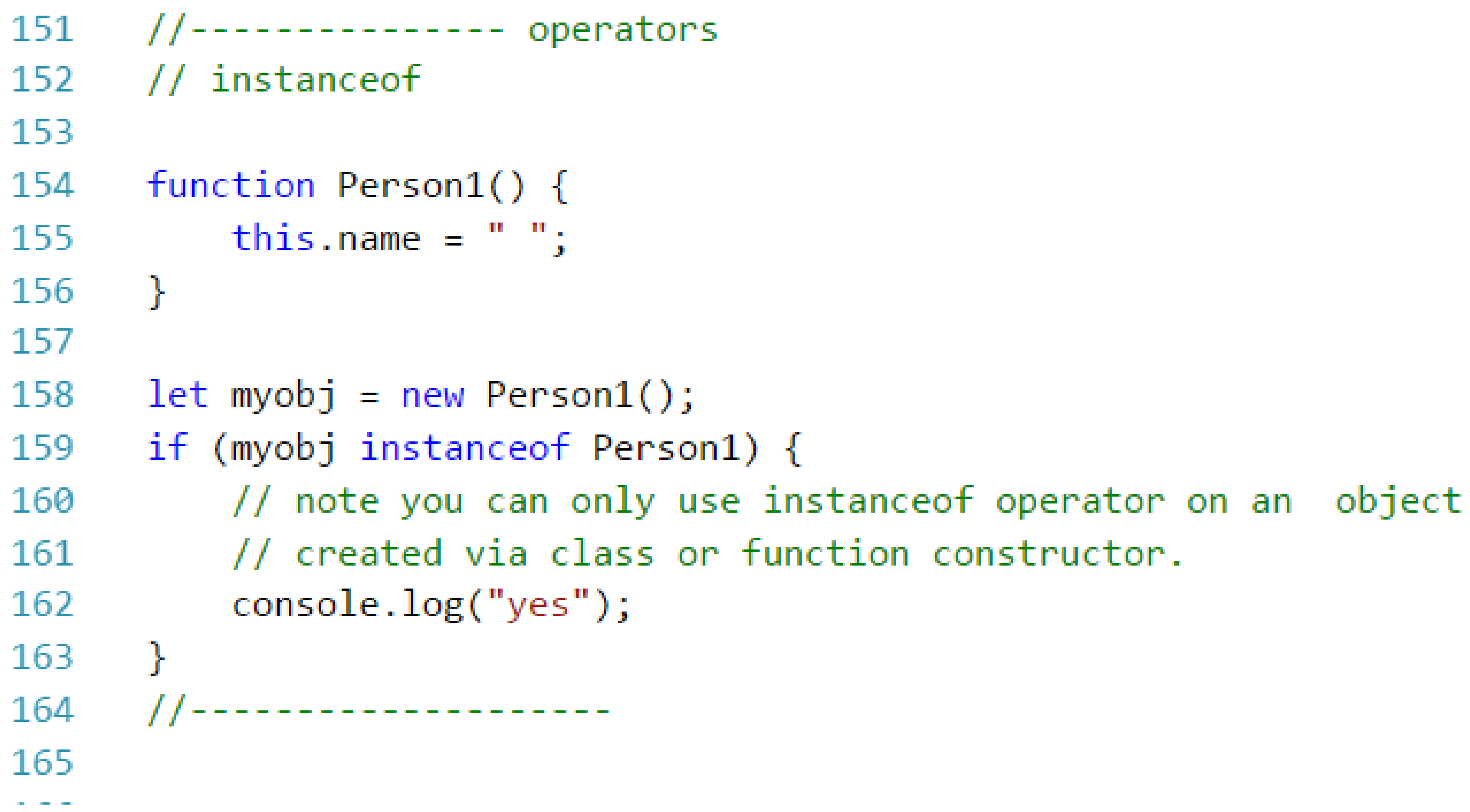
```javascript
class Person2{

    constructor(firstName,lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }

    fullName(){
        return "From Person2 " + this.firstName + " " + this.lastName;
    }
}
const p2 = new Person2('george',"jones");
console.log(p2.fullName());

// We can refer to object properties as either
// data properties - i.e. they are either a primitive value or object reference
// function properties - i.e. they refer to a function or perhaps in a class we describe them as methods

// as brenden Hike said - the protoype should be used to shared function across instances
// and to share immuntal data like constants - but of course this immutalability is not enforced


```

__proto__ & prototype



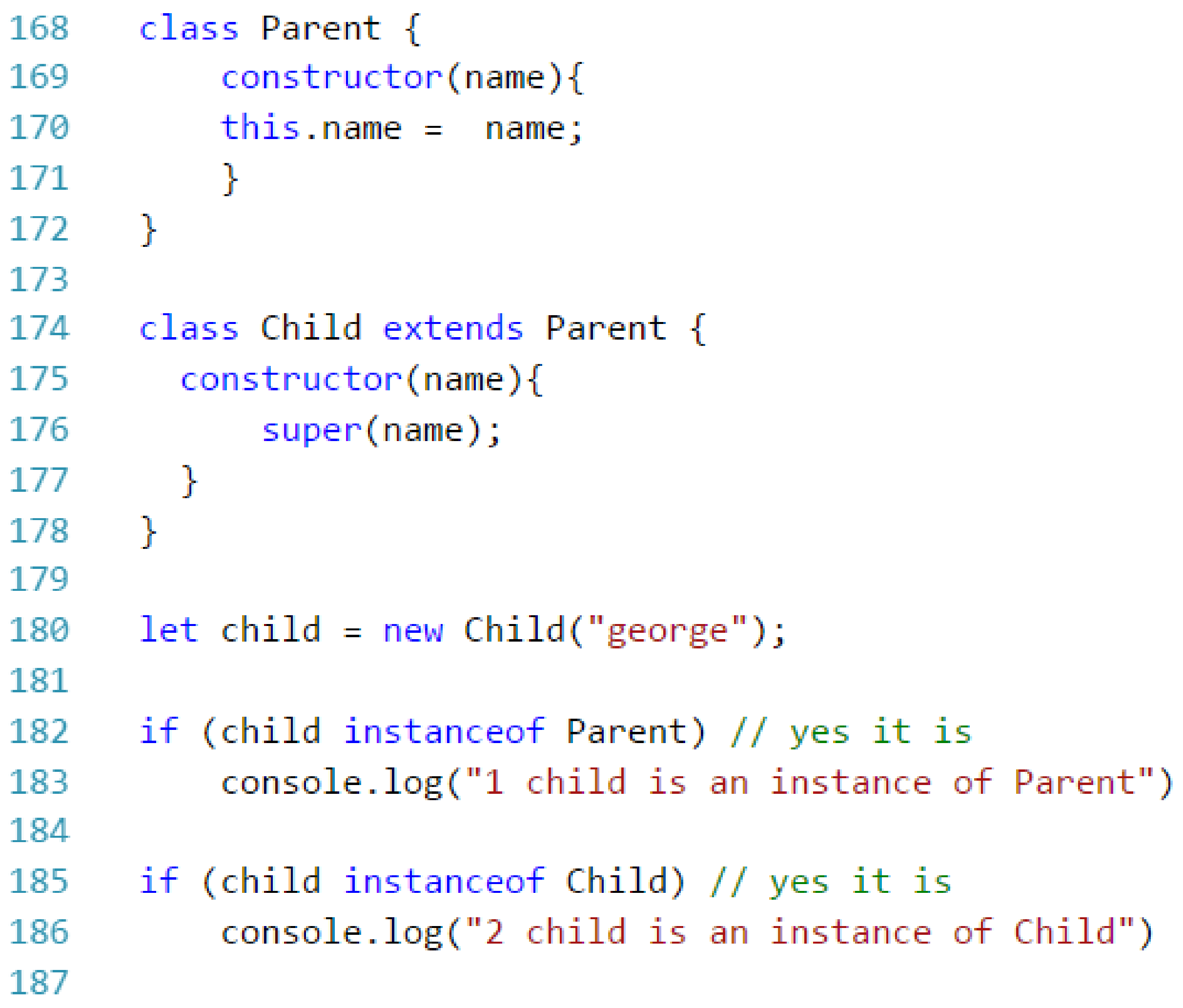
```
// does p2's __proto__ point to the Person2's prototype
console.log(p2.__proto__ === Person2.prototype); //true

// so i.e. if we look at Person2's ( __proto__ that would be function() )
// and (prototype would be Object)
// __proto tells us what we are inheriting from
// prototype is used to construct our object, when we use the new keyword etc

console.log(Object.getPrototypeOf(p2) === Person2.prototype);  //true
console.log(Object.getPrototypeOf(Person2) === Person2.prototype); //false


```

instanceof



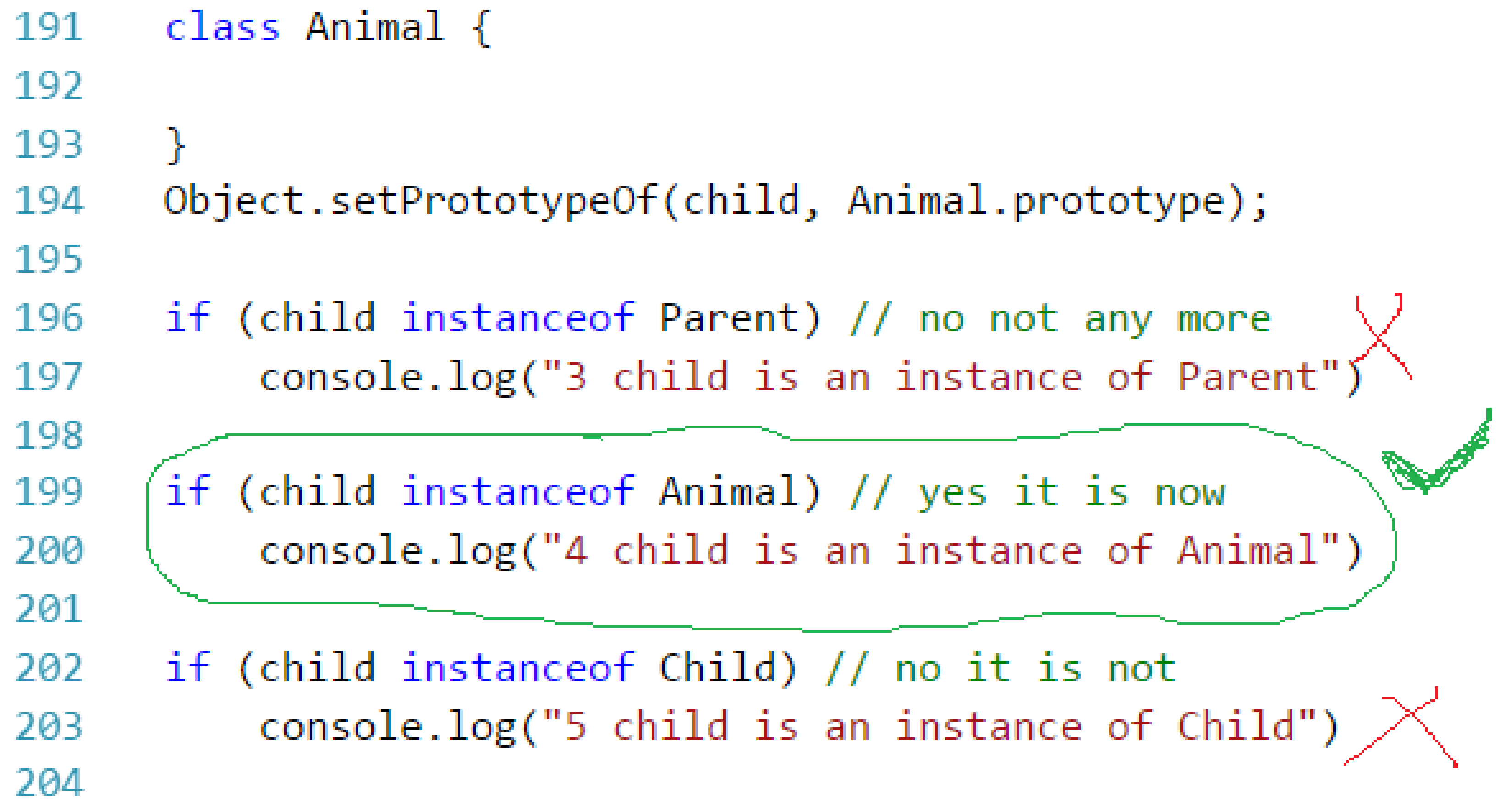
```javascript
//--------------- operators
// instanceof

function Person1() {
    this.name = " ";
}

let myobj = new Person1();
if (myobj instanceof Person1) {
    // note you can only use instanceof operator on an  object
    // created via class or function constructor.
    console.log("yes");
}
//--------------------

```

Subclass extend a Parent class



```
class Parent {
    constructor(name){
    this.name =  name;
    }
}

class Child extends Parent {
  constructor(name){
      super(name);
  }
}

let child = new Child("george");

if (child instanceof Parent) // yes it is
    console.log("1 child is an instance of Parent")

if (child instanceof Child) // yes it is
    console.log("2 child is an instance of Child")
```

Object.setPrototypeOf

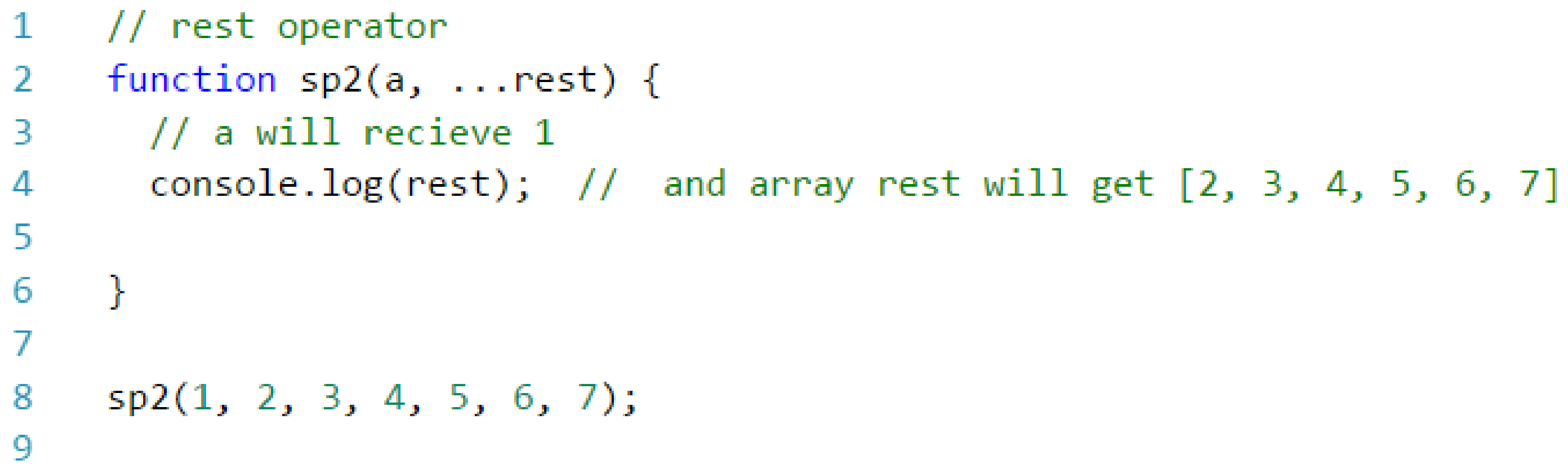


```
class Animal {

}
Object.setPrototypeOf(child, Animal.prototype);

if (child instanceof Parent) // no not any more
    console.log("3 child is an instance of Parent")

if (child instanceof Animal) // yes it is now
    console.log("4 child is an instance of Animal")

if (child instanceof Child) // no it is not
    console.log("5 child is an instance of Child")
```

Destructing

```javascript
const person = {

    name        :   "george",
    age         :   20,
    interests   :   "programming"

};

// brackets on the left of the assigment operator means destructing
let { name , age, interests } = person;

// we can list any order or number of prop keys to use
let { interests, age } = person;

// personsName is the variable name we use for key of name
let { name : personsName, age, interests } = person;

// we can have default values
let { age, interests, iDontExist = "I do now" } = person;

// lets print out using a back ticks i.e. interpolation with a dollar in front
//of brackets containing our variable
console.log( ` age= ${age } , interests= ${ interests } , iDontExist= ${ iDontExist }  `)

//const { prop1, prop2, prop3= " A default value"  } = theobject;

```

ES6 … Rest operator

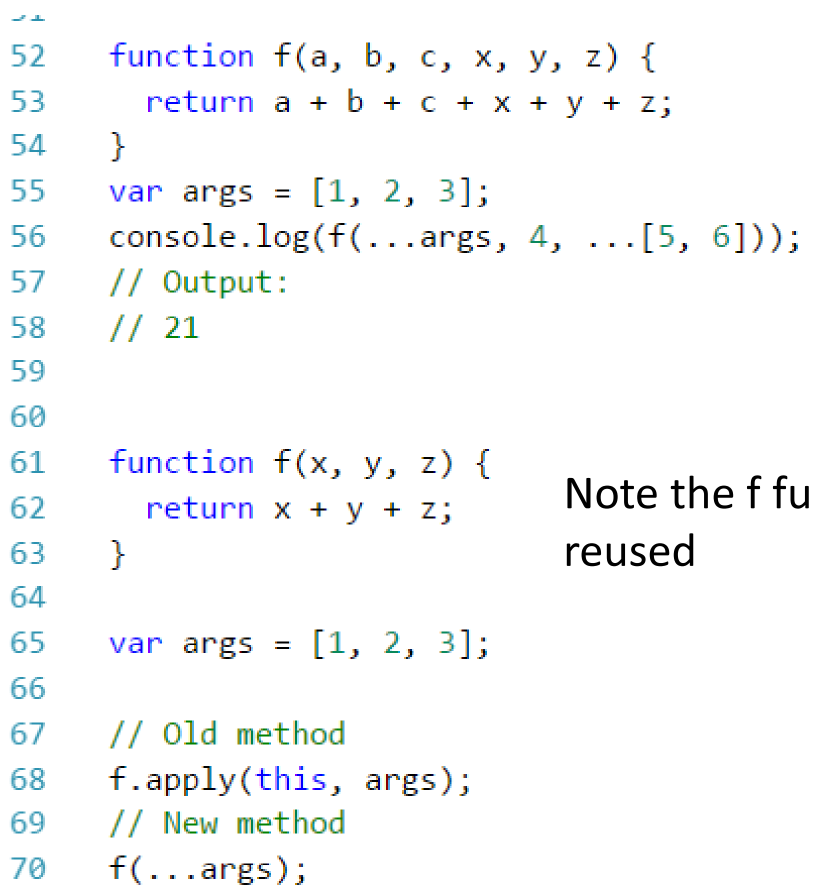


```
// rest operator
function sp2(a, ...rest) {
  // a will recieve 1
  console.log(rest);  //  and array rest will get [2, 3, 4, 5, 6, 7]

}

sp2(1, 2, 3, 4, 5, 6, 7);

```

…Spread operator

```
let boys = ['jim', 'james', 'george'];

function sp(a, b, c) {
    // any parameters passed i.e. more than 3 are placed
    //  into the function's arguments variable
    console.log(a, b, c)
}
sp(...boys);

// using spread to expand the boys array into Girls name array
let girls = ['sally', 'ann', ...boys, 'linda'];
// i.e. now girls = ['sally', 'ann','jim', 'james', 'george', 'linda'];
console.log(girls);


```

And more …Spread



```
function f(a, b, c, x, y, z) {
  return a + b + c + x + y + z;
}
var args = [1, 2, 3];
console.log(f(...args, 4, ...[5, 6]));
// Output:
// 21


function f(x, y, z) {
  return x + y + z;
}

var args = [1, 2, 3];

// Old method
f.apply(this, args);
// New method
f(...args);
```

Note the f function has abeen reused

this ☺

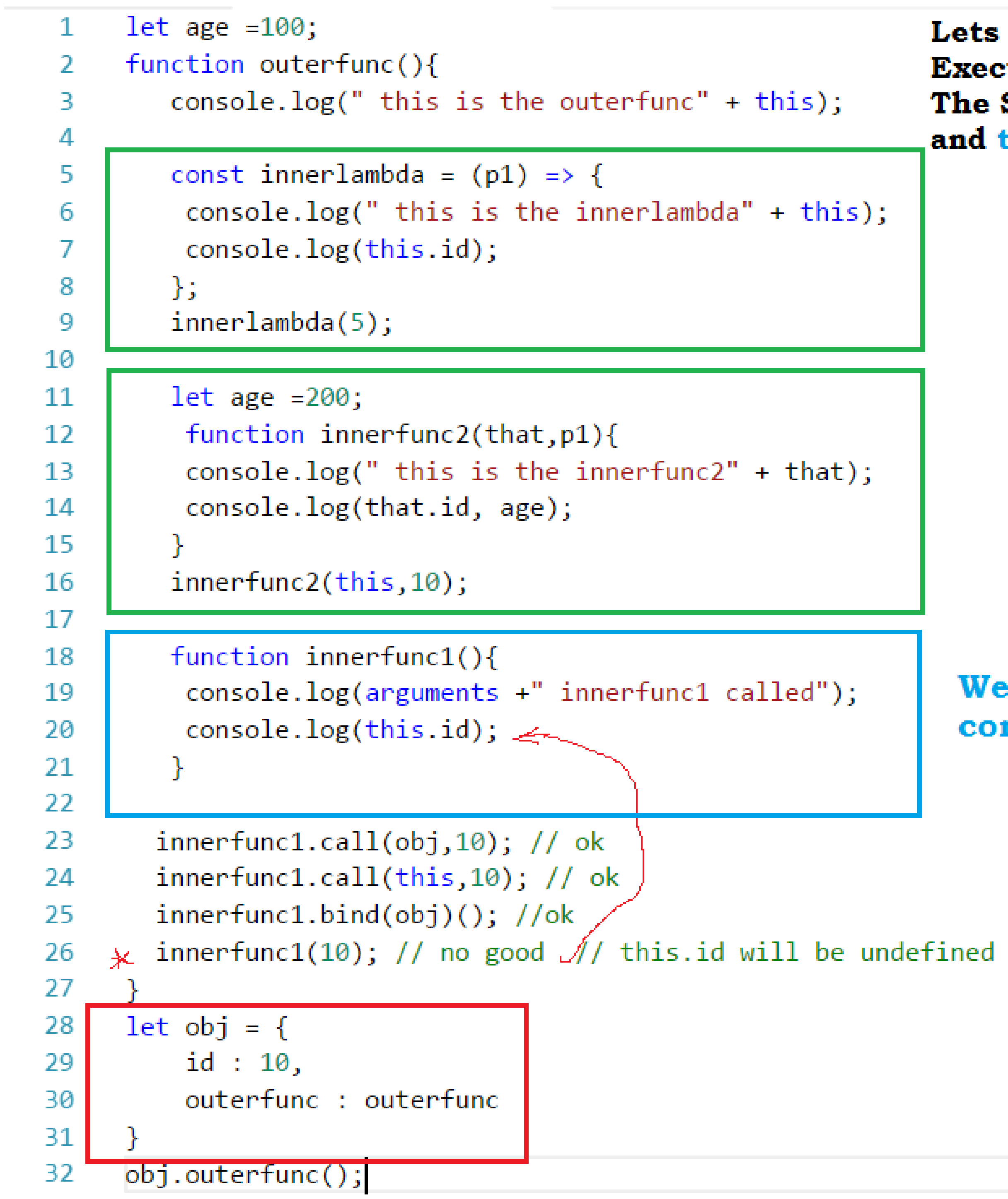
Creation Stage (when the function is called, but before it executes any code inside)

- Create the Scope Chain.
- Create variables, functions and arguments.
- Determine the value of "this".

Activation / Code Execution Stage:

- Assign values, references to functions and interpret / execute code.

Lets look at Execution Context, The Scope Chain, and this

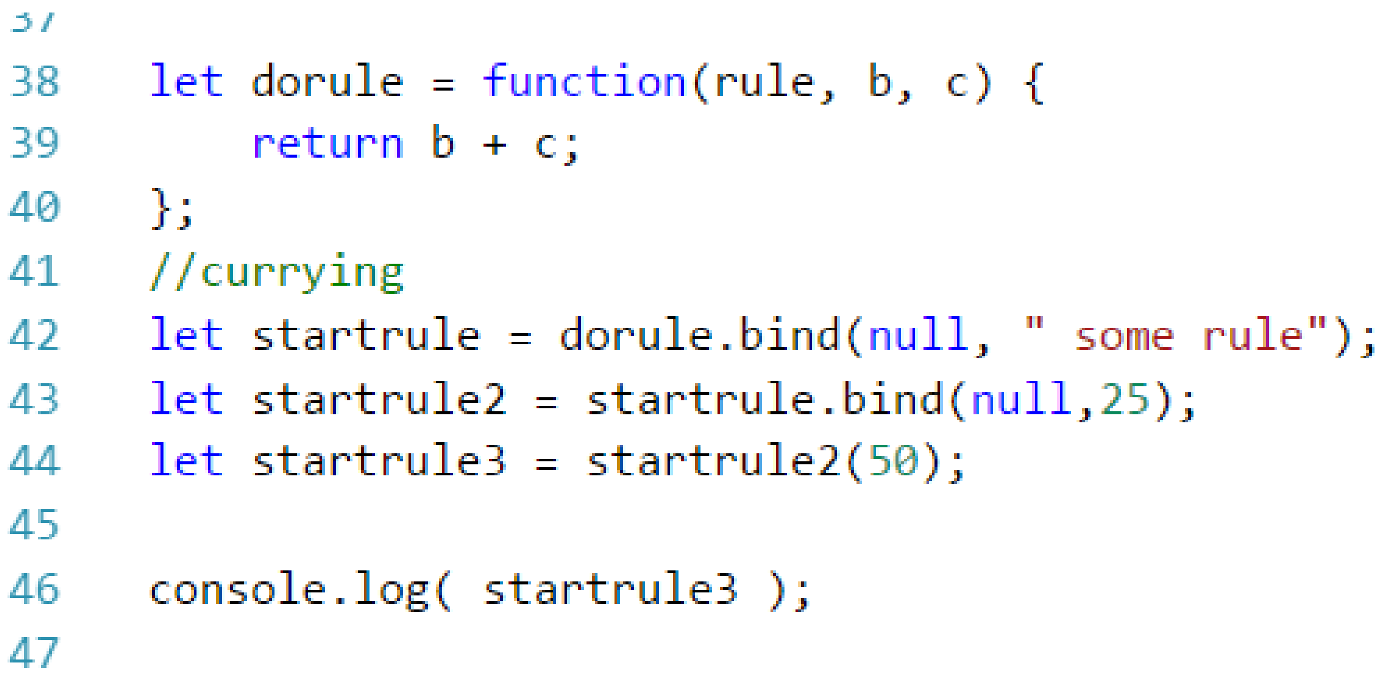


```
let age =100;
function outerfunc(){
   console.log(" this is the outerfunc" + this);

   const innerlambda = (p1) => {
    console.log(" this is the innerlambda" + this);
    console.log(this.id);
   };
   innerlambda(5);

   let age =200;
    function innerfunc2(that,p1){
    console.log(" this is the innerfunc2" + that);
    console.log(that.id, age);
   }
   innerfunc2(this,10);

   function innerfunc1(){
    console.log(arguments +" innerfunc1 called");
    console.log(this.id);
   }

  innerfunc1.call(obj,10); // ok
  innerfunc1.call(this,10); // ok
  innerfunc1.bind(obj)(); //ok
  innerfunc1(10); // no good // this.id will be undefined
}
let obj = {
    id : 10,
    outerfunc : outerfunc
}
obj.outerfunc();
```

We are only concerned here

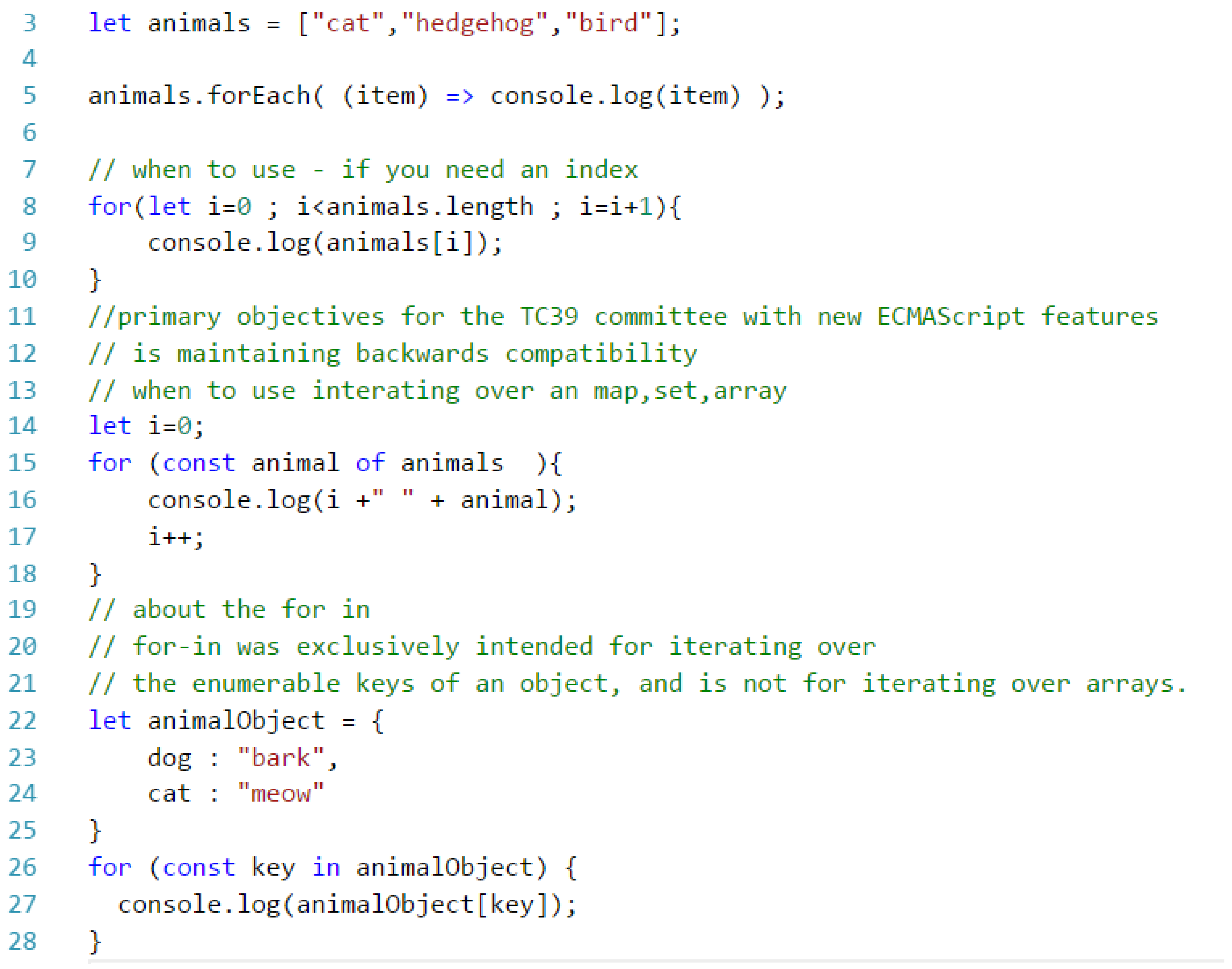
On the subject of bind

- We can do currying and partial application
- i.e.



```
let dorule = function(rule, b, c) {
    return b + c;
};
//currying
let startrule = dorule.bind(null, " some rule");
let startrule2 = startrule.bind(null,25);
let startrule3 = startrule2(50);

console.log( startrule3 );

```

ForEach, For in , (for of es6/2015) - iterations



```javascript
let animals = ["cat","hedgehog","bird"];

animals.forEach( (item) => console.log(item) );

// when to use - if you need an index
for(let i=0 ; i<animals.length ; i=i+1){
    console.log(animals[i]);
}
//primary objectives for the TC39 committee with new ECMAScript features
// is maintaining backwards compatibility
// when to use interating over an map,set,array
let i=0;
for (const animal of animals  ){
    console.log(i +" " + animal);
    i++;
}
// about the for in
// for-in was exclusively intended for iterating over
// the enumerable keys of an object, and is not for iterating over arrays.
let animalObject = {
    dog : "bark",
    cat : "meow"
}
for (const key in animalObject) {
  console.log(animalObject[key]);
}
```

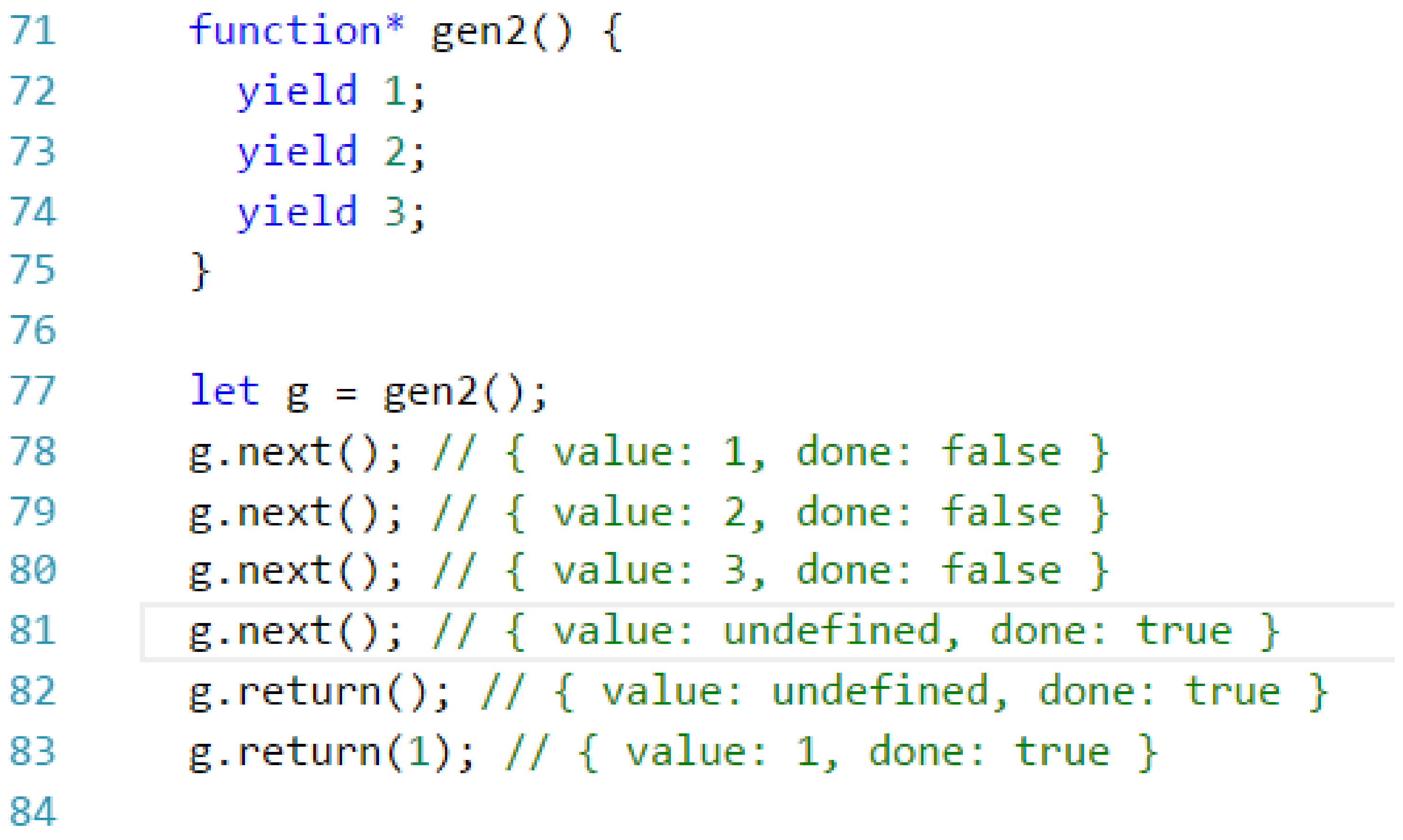
Function Generators

```
// yield returns an object as such {value: the_value_tobe_yield, done: true}.
//  when you have either yield'd all the values or return then
//  the done flag will be set to true - calls after done is true will have a value of undefined

function* generator(i) {
    yield i;  // starts here - yield returns this value and advances to the line

    yield i + 10; // next time it's called it starts here yield returns a value and advances to the next

    for (let j = i; j <= (i + 40); j = j + 10) {
      yield j; // return this value and
    }
    yield 10000;
    let count=0;
    while (true) {
      count = count+1;
      yield function () { return Math.floor((Math.random() * 52) + 1); }
      console.log("running ...");
      if(count> 2){
          return 50000; // we could also end the yield completely and return
      }

    }
  }
```

The results of calling the generator function

```
var gen = generator(10);

var value1 = gen.next().value; // expected output: 10
var value2 = gen.next().value; // expected output: 20
var value3 = gen.next().value; // expected output: 10
var value4 = gen.next().value; // expected output: 20
var value5 = gen.next().value; // expected output: 30
var value6 = gen.next().value; // expected output: 40
var value7 = gen.next().value; // expected output: 50
var value8 = gen.next().value; // expected output: 10000
var value9 = gen.next().value; // expected output: 1 to 52
var value10 = gen.next().value; // expected output: 1 to 52
var value11 = gen.next().value // expected output: 1 to 52
var value12 = gen.next().value; // expected output: 50000

console.log(value1); // expected output: 10
console.log(value2); // expected output: 20
console.log(value3); // expected output: 10
console.log(value4); // expected output: 20
console.log(value5); // expected output: 30
console.log(value6); // expected output: 40
console.log(value7); // expected output: 50
console.log(value8); // expected output: 10000
console.log("first rnd = " + value9()); // expected output: 1 to 52
console.log("second rnd = " + value10()); // expected output: 1 to 52
console.log("third rnd = " + value11()); // expected output: 1 to 52
console.log(value12); // expected 50000
```

Another example



```
function* gen2() {
  yield 1;
  yield 2;
  yield 3;
}

let g = gen2();
g.next(); // { value: 1, done: false }
g.next(); // { value: 2, done: false }
g.next(); // { value: 3, done: false }
g.next(); // { value: undefined, done: true }
g.return(); // { value: undefined, done: true }
g.return(1); // { value: 1, done: true }
```

Passing items into a Generator call

```
function* numberGuess() {
  console.log("starting guessing game");
  const reply = yield 'Question is number > 100';    // {done:false, value ='Question is number > 100'}
  console.log(reply);

  console.log("ending guessing game");
  if (reply !== 'yes') return 'Wrong'                 // {done : true, value = 'Wrong' }
  return 'Correct';                                   // {done : true, value = 'Correct'}

}


const iter = numberGuess();// initialize our generator iter variable instance.
// Iterator .next yields question - it return an object with a boolean done flag and value
const question = iter.next().value; //we are only interested in the value-

console.log(question);
const answer = iter.next('yes').value; // Pass reply back into generator
console.log(answer);
```

Make you own Iterator

```
function makeIterator(array) {
    var nextIndex = 0;

    return {
       next: function() {
           return nextIndex < array.length ?
               {value: array[nextIndex++], done: false} :
               {done: true};
       }
    };
}
```

Once initialized, the `next()` method can be called to access key-value pairs from the object in turn:

```
var it = makeIterator(['yo', 'ya']);
console.log(it.next().value); // 'yo'
console.log(it.next().value); // 'ya'
console.log(it.next().done);  // true
```

Make you own Iterator

```
// custom iterator example

let addressBook = {
    bookList : ["Java","C#", "Kotlin","Python","F#", "Javascript","Typescript","Scala"]
};

addressBook[Symbol.iterator] = function(){

    let thiscontext=this;
    return {
        next(){
            if(thiscontext.bookList.length > 0){
               return {value:thiscontext.bookList.shift(),done:false};
            }
            else{
               return {value:thiscontext.bookList.shift(),done:true};
            }
        }
    }
}


 for(let title of addressBook){

     console.log(title);
 }
```

Async/Await

```
const delay = (seconds) => {
    return new Promise(
        resolve => setTimeout( resolve,seconds * 1000 )
    )
};

const countToFive = async() => {
    console.log(' 0 Seconds ');
    await delay(1);
    console.log(' 1 Seconds ');
    await delay(1);
    console.log(' 2 Seconds ');
    await delay(3);
    console.log(' 5 Seconds ');
    return new Promise((resolve) => {
                resolve("resolved!!");
            });

};

countToFive().then(
    (text) => {
        console.log('outside: ' + text)
        console.log("the end");
     },
    (err)  => { console.log(err) }
 )
```

The End for Now

- I will update the slides to include a lot of other items, this was just a small selection to aid the discussion.

- Thanks – See you next time at the
  Limerick Ai Software Development meetup