JAVASCRIPT2

Visual Studio Code

1.27.2

Free, open source code editor with PowerShell terminal developed by Microsoft for Windows, Linux and MacOS that supports a number of programming languages, such as JavaScript, Java and C#

Node.js

8.12.0

JavaScript runtime built on Chrome's V8 JavaScript engine
Npm: Package manager for JavaScript and the world's largest software registry
Babel: Tool chain used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments
CommonJS: Project with the goal of specifying an ecosystem for JavaScript outside the browser
WebPack: Static module bundler for modern JavaScript applications, that takes dependencies, generates a dependency graph, packages resources and assets and creates bundles

ECMASCRIPT

ECMAScript6 (2015)
Let / Const / Parameter values / Arrow functions / New array, number and global methods / Promises / Classes / Collections / Modules
ECMAScript7 (2016)
Typed objects / Object observation / Async functions
ECMAScript8 (2017)
Async / Await

FEATURES & SYNTAX

Var

Function scoped

Const

Block scoped
No hoisting
Immutable

Let

Block scoped
No hoisting

Use let and const instead of var

Classes & Inheritance

Keywords: class / new / extends

```
class Person {
    constructor(name) { this.name = name }
    hello() { return 'Hello, My name is ' + this.name + '.' }
}
class Actor extends Person {
    hello() { return super.hello() + ' I am an actor.' }
}
const actor = new Actor('Jim Wilson')
console.log(actor.hello());
```

Use classes instead of constructor functions

Arrow functions

Also called "fat arrow" functions / Work much like Lambdas

Arrow functions are anonymous and have shorter syntax than a regular functions, change the way this binds and the return keyword is implicit

```
// ES5
var es5func = function (a, b) { return a * b }
// ES6
const es6func = (a, b) => { return a * b };
const es6funcDefaultParameters = (a = 3, b = 3) => { return a * b };
const es6funcSingleParameter = a => { return a * a };
const es6funcNoReturn = a => a * a;
```

Not intended to replace regular functions

Modules

ES6 is the first time that JavaScript has built-in modules and own modules can be created

Import statement is used to export bindings for other modules

ES6 Module Export Syntax

```
export default something
```

CommonJS Module Export Syntax

```
exports.something = s;
module.exports = { something : s }
```

Import statement is used to import bindings which are exported by another module

Importing is done via the (import ... from ...) construct

ES6 Module Import Syntax

```
import something from "./someModule";
```

CommonJS Module Export Syntax

```
const someModule = require('./someModule');
someModule.something
```

Arrays

Map

Creates a new array with the results of calling a provided function on every element in the calling array

```
const numbersHalved = numbers.map(value => value / 2);
```

Filter

Creates a new array with all elements that pass the test implemented by the provided function

```
const numbersAbove5 = numbers.filter(value => value > 5);
```

Reduce

Applies a function against an accumulator and each element in the array to reduce it to a single value

```
const numbersSum = numbers.reduce((sum, value) => sum + value);
```

Join

Joins elements of an array into a string, and returns the string

```
const numbersJoined = numbers.join(" / ");
```

Spread

Performs all of the array manipulations that previously required push, concat, or slice methods

```
const numbersSpread = [...numbers, ...numbers];
```

CallBack functions

A callback function is a function that is passed as a parameter to another function to be executed inside it

```
const callback = (message) => { console.log(message); }
const func1 = (text, cb) => {
    cb(text);
};
func1("This is some text!!!", callback);
```

<u>“Callback Hell” / “The Pyramid of Doom”</u>

Synchronizing asynchronous tasks is difficult with callback functions

Many levels of nested indentation

```
const callbackHell = () => {
    console.log('1. First function sets up second function');
    setTimeout(function () {
        console.log('2. Second function sets up third function');
        setTimeout(function () {
            console.log('3. Third function finishes');
        }, 2000);
    }, 2000);
};
callbackHell();
```

Promises

Do not remove the use of callbacks / Make chaining of functions straightforward / Simplify the code

Represents the completion or failure of an asynchronous action and its value
A promise has 3 states…

- Pending: Final value not available
- Fulfilled: Final value available
- Rejected: Rejection reason

Creating a promise

```
const promise = new Promise((resolve, reject) => {
    return resolve({ message: 'Resolved!!!' });
    //return reject( new Error('Rejected!!!') );
});
```

Execute promise

```
promise1
.then((resolve) => console.log(resolve.message))
.catch((reject) => console.log(reject.message));
```

Eliminates the famous “Callback Hell”

```
const callbackHellPromises = () => {
    new Promise((resolve) => {
        console.log('1. First function sets up second function!!!');
        setTimeout(resolve, 2000);
    })
    .then(() => new Promise((resolve) => {
        console.log('2. Second function sets up third function!!!');
        setTimeout(resolve, 2000);
    }))
    .then(() => new Promise(() => {
        console.log('3. Third function finishes!!!');
    }))
    .catch(error => {
        console.log('Error occured!!!');
    })
};
callbackHellPromises();
```

Async / Await

JavaScript is asynchronous by default
async and await statements are syntactic sugar on top of promises in JavaScript
Makes it possible to write promise based code as if it were synchronous, but without blocking the main thread
Keyword async can be placed before functions and means functions will return a promise
Keyword await makes JavaScript wait until promise settles and returns the result and it can only be used inside async functions
Code is much cleaner / Error handling is much simpler and it relies on try/catch / Debugging is much simpler

```
const asyncAwait = async () => {
    try
    {
        await new Promise((resolve) => {
            console.log('1. First function sets up second function...');
            setTimeout(resolve, 2000);
        });
        await new Promise((resolve) => {
            console.log('2. Second function sets up third function...');
            setTimeout(resolve, 2000);
        });
        await new Promise((resolve) => {
            console.log('3. Third function finishes...');
        });
    }
    catch(error)
    {
        console.log('Error occured!!!');
    }
};
asyncAwait();
```

Closures
Nested functions
A closure is the combination of a function and the environment within which that function was declared
Closures are useful because they make it possible to associate some data with a function that operates on that data
One powerful use of closures is to use the outer function as a factory for creating related functions

```
const add = (function () {
    let counter = 0;
    return function() {counter += 1; return counter}
})();
console.log("ADD: " + add());
```

Emulating private methods with closures
Defining public functions that can access private functions and variables is known as the module pattern

```
const counterModulePattern = () => {
    let counter = 0;
    return {
        increment: () => { counter++; },
        decrement: () => { counter--; },
        value: () => { return counter; }
    }
};
const counter1 = counterModulePattern();
const counter2 = counterModulePattern();
console.log(counter1.value());
counter1.increment();
counter1.increment();
console.log(counter1.value());
counter1.decrement();
console.log(counter1.value());
console.log(counter2.value());
```

Do not unnecessarily create nested functions if closures are not needed, as it will negatively affect performance

Scopes

Different scopes exist
Global / Function / Local

This
In most cases, the value of this is determined by how a function is called
Can't be set by assignment during execution, and it may be different each time the function is called

Global object scope
Outside of any function
this refers to the global object
Regular function scope
this scope refers to nearest function
Arrow function scope
Doesn't have own this binding
this scope is inherited from the context
Constructor scope
this is bound to the new object being constructed

Object method scope

this is set to the object the method is called on

DOM event handler

this is set to the element the event fired from

Inline event handler

this is set to the DOM element on which the listener is placed

```
const globalScope = this;
console.log("GlobalScope: " + this);

const regularFunctionScope = function () { return this; };
console.log("RegularFunctionScope: " + regularFunctionScope());

const arrowFunctionScope = () => { return this; };
console.log("ArrowFunctionScope: " + arrowFunctionScope());

class Obj {
    constructor() { this.theClass = this; }
    func() { return this.theClass; }
    check() { return true; }
}
const obj = new Obj();
console.log("ConstructorScope: " + obj.theClass.check());
console.log("MethodScope: " + obj.func().check());
```

Bind / Call / Apply

3 methods used to control invocation of functions

Set which specific object will be bound to this when function is invoked

Bind: Returns a new function

```
const person1 = { name:"Steven" };
const greeting1 = function(place){ return "welcome " + this.name + ", to " +
place; };
const person1Greeting = greeting1.bind(person1, "right here");
console.log(person1Greeting());
```

Call: Invokes function

```
const person2 = { name:"Stephen" };
const greeting2 = function(place){ return "welcome " + this.name + ", to " +
place; };
const person2Greeting = greeting2.call(person2, "right here");
console.log(person2Greeting);
```

Apply: Invokes function with array argument

```
const person3 = { name:"Stephan" };
const greeting3 = function(place){ return "welcome " + this.name + ", to " +
place; };
const person3Greeting = greeting3.apply(person3, ["right here"]);
console.log(person3Greeting);
```

JAVASCRIPT PROJECTS

npm install

Fetch project dependencies

npm run build

Build project

npm start

Start project

SIMPLESPA

Frontend project / Implement find joke and add joke

/public	index.html / style.css	Add html / Add css
/src	index.js / jokes.js / style.css	Add js

JSONSERVER

Backend REST project

AJAX

XMLHTTPREQUEST

“Old way of doing ajax requests”
Uses the XMLHTTPRequest object

FETCH

“New way of doing ajax requests”
Built on the promise object which greatly simplifies the code, especially if used in conjunction with async / await

ErrorHandling

Errors in json format from REST api can be extracted with Javascript
Catch / Status codes / json object with properties

SINGLE PAGE APPLICATIONS

Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update the page as the user interacts with the app
Via JavaScript SPAs use AJAX and DOM to create Web apps without constant page reloads on the client side
Client code (Bundled SPA) can be hosted on one server and server code (REST API) on another server
Different servers communicating via JavaScript and AJAX leads to problems with the same origin policy, can be solved with cross origin resource sharing

SAME ORIGIN POLICY / CORS

Connection between different servers via AJAX can be blocked

- HTTP Header can be used to allow external requests
- HTTP Reponse filter can be used to allow external requests
- HTTPURLConnection can be used to proxy external server requests to own REST api

DEPLOYMENT

Deploy backend to Digital Ocean
Deploy frontend to Surge.sh

```
npm run build
npm install -g surge
surge --project ./build --domain A_DOMAIN_NAME.surge.sh
        email / password
```