Table of Contents

General Concepts

Syntax parsers

A program that reads your code and determine what it does and if its grammar is correct.

Lexical environments

Where something sits physically in the code you write.

Place where the parts of your code sits, is important.

Execution contexts

A wrapper to help manage the code that is running.

Name/Value pair

A name which maps to a unique value

```
Address: '100 main str'
```

Object

A collection of name value pairs.

```
Address:
        {
        Street:'Main',
        Number:100,
        Apartment:{
            Floor: 3,
            Number:301
        }
    }
```

Global Environment and The Global Object

Global = not inside a function

Execution Context

- Global Object = Window Object in browsers.
- 'this' = Window Object in browsers.
- Link to the Outer Environment
- Your code

Creation Phase:

- Global Object
- 'this'
- Outer Environment
- "Hoisting" Setup Memory Space for Variables and functions
    - Functions are set to memory entirely (name and code).
    - Variables are set to a placeholder called *undefined*. Because the JS engine won't know the space until it executes it. All variables are initialy set to *undefined*.

```
b();
    console.log(a);

    var a = 'Hello World'

    function   b(){
        console.log('Called b');
    }
    // Returns
    > 'Called b'
    > undefined
```

Execution Phase

Executes the code line by line.

Single Threaded Synchronous Execution

One at a time, in order.

Function invocation

Execute, run a function.

Execution Stack

When a function is called a new execution context is created and put at the top of the execution stack.

```
function b() {}
    function a() { b()}
    a();
```

The execution stack is.

- b() Execution Context (create and execute)
- a() Execution Context (create and execute)
- Global Execution Context (created and code executed)

Variable environments.

Where the variables live, and how they relate to each other in memory.

Scope Chain

Functions use outer environments to find variables that are not in its execution contexts.

```
function b() {
        console.log(myVar);
    }

    function a() {
        var myVar = 2;
        b();
    }

    var myVar = 1;
    a();
    //Returns
    > 1
    //'b' outers environment is the global environment.
```

```
function a() {

        function b() {
            console.log(myVar);
        }

        b();
    }

    var myVar = 1;
    a();

    //Returns
    > 2
    //'b' outers environment is the 'a' environment.
```

Scope

Scope is where a variable is available in your code

ES6 let

Block scoping: define a variable that it is only available in its block(if block, for loop block...), and can not be used until is declared.

Asynchronous

More than one at a time. Event Queue is processed after the execution stack is empty.

```
// long running function
    function waitThreeSeconds() {
        var ms = 3000 + new Date().getTime();
        while (new Date() < ms){}
        console.log('finished function');
    }
    function clickHandler() {
        console.log('click event!');
    }

    // listen for the click event
    document.addEventListener('click', clickHandler);


    waitThreeSeconds();
    console.log('finished execution');
    //Returns if clicked
    > finished function
    > finished execution // The execution task is empty
    > click event
```

3 Types and Operators

Dynamic Typing

Variable types are figured out while the code is running.

Primitive Types

Data that represent a single value (not an object)

1. `undefined`: lack of exixtence (do not set to this value)
2. `null`: lack of exixtence
3. `boolean`
4. `number`: Floating point
5. `string`
6. `symbol`: ES6

Operators

A special function written differently as common functions.

- Infix notation `3 + 4` used in javascript
- Prefix notation `+3 4`
- Postfix notation `3 4+`

Precedence and Associativity

Precedence: Which Operator function gets called first.

Associativity: What order Operator functions get called: left or right or right to left

```
var a = 2, b = 3, c = 4
```

```
    a = b = c
    //all equals to 4
```

Operator precedence and Associativity table

Coercion

Converting a value to one type to another

```
var a = 1 + '2';
    // 12
```

Comparison Operators

Use `===` 99% of the time unless you want to coarce and you know what you expect of the coarce.

Equality comparisons and sameness

Default values

```
function greet(name){
        name = name || '<Your name here>'
        console.log('Hello ' + name)
    }

    greet();
```

Objects and Functions

Objects and Dots

```
var person = new Object();

    person["firstname"] = "Tony";

    var firstNameProperty = "firstname";

    //Computed member access
    console.log(person[firstNameProperty]);

    //Member access
    console.log(person.firstname);

    // An object inside another object (subobjects).
    person.address = new Object();
    // Left to right Associativity. Person then address then street.
    person.address.street = "111 Main St.";

    console.log(person.address.street);

    //Same thing
    console.log(person["address"]["street"]);
```

Object literals

Shorthand using curly braces.

```
var person = {
        firstname: 'Tony', // They can be initialized
        address: {
            street: '111 Main St.',
        }
    };
```

```
var Tony = {
        firstname: 'Tony',
        address: {
            street: '111 Main St.',
        }
    };

    function greet(person) {
        console.log('Hi ' + person.firstname);
    }

    greet(Tony);

    // Creating an object on the fly
    greet({
        firstname: 'Mary',
        lastname: 'Doe'
    });

    // Add new properties on the fly
    Tony.address2 = {
        street: '333 Second St.'
    }
```

NameSpace

A container for variables and functions.

Faking name spacing Containt objects, methods and properties inside a container object.

```
var greet = 'Hello!';
    var greet = 'Hola!';

    console.log(greet); // Returns: Hola!

    var english = {};
    var spanish = {};

    english.greet = 'Hello!';
    spanish.greet = 'Hola!';

    console.log(english); // Returns:  Object {greet: "Hello!"}
```

JSON and Object literals

A JSON is an Object

An Object could not be a JSON

Properties have to be wrapped in cuotes

No functions as values

JSON

```
{
        "firstname": "Mary",
        "isAProgrammer": true
    }
```

OBJECT

```
var objectLiteral = {
        firstname: 'Mary',
        isAProgrammer: true
    }

    console.log(JSON.stringify(objectLiteral));

    var jsonValue = JSON.parse('{ "firstname": "Mary", "isAProgrammer": true }');
```

Functions and Objects

First class functions. They can:

- Assign variables to have a value that is a function
- Pass functions as parameters
- Create functions on the fly

Functions are special Objects.

They can have

- Primitives
- Objects
- Other functions

Only for functions * **Name**: Can be optional: *annonymous* * **Code**: is invocable.

```
function greet() {
        console.log('hi');
    }

    greet.language = 'english';
    console.log(greet.language); // 'english'
```

Function Statements and Expression

Expression: unit of code that results in a value

Statement: Does not return a value ie. `if statement`

Function statement

```
//This function has a name, but is not assigned
    function greet() {
        console.log('hi');
    }

    greet();
```

Function expression

```
//This function has no name, but is assigned to a variable
    var anonymousGreet = function() {
        console.log('hi');
    }


    anonymousGreet();
```

Pass a function to a function

```
function log(a) {
       a();
    }

    log(function() {
        console.log('hi');
    });
```

By Value vs. by Reference

Primitive values are passed **by value**, are copied in a new memory address.

Objects are passed **by reference**, a new pointer is created pointing to the same address.

= Operator sets up a new memory space (new address)

```
var c = { greeting: 'hi' };
    var d;
    d = c; // d and c point to the same address

    c = { greeting: 'howdy' }; / // d and c do not point anymore to the same
address
    console.log(c);
    console.log(d);

    //Returns
    > howdy
    > hi
```

Objects, Functions and 'this'

Calling `this` from a function, will point always to the `Window` object.

```
function a() {
        console.log(this);
    }

    var b = function() {
        console.log(this);
    }

    a();
    b();

    //Returns
    > Window
    > Window
```

Object literal with methods. (functions inside an objects are called methods)

```
var c = {
        name: 'The c object',
        log: function() {
            this.name = 'Updated c object';
            console.log(this);  // Point to the 'c' object
        }
    }

    c.log(); // Will show the c object where in its 'name' property will be
'Updated c object';
```

Creating a function inside an object (an internal function) and using `this` inside it, will point to the `Window` object instead to the object.

To fix this we can use the `self` pattern.

```
var c = {
        name: 'The c object',
        log: function() {
            var self = this;

            self.name = 'Updated c object';
            console.log(self);

            var setname = function(newname) {
                this.name = newname; // Here 'this' points to the 'Window' object
instead of to 'c'
                self.name = newname; // Here 'this' points to 'c'
            }
            setname('Updated again! The c object');
            console.log(self);
        }
    }

    c.log();
```

Arrays

Arrays can hold anything and we can use them without any problem.

```
var arr = [
        1,
        false,
        {
            name: 'Tony',
            address: '111 Main St.'
        },
        function(name) {
            var greeting = 'Hello ';
            console.log(greeting + name);
        },
        "hello"
```

```
    ];

    console.log(arr);
    arr[3](arr[2].name); // 'Hello Tony'
```

'arguments' and SPREAD

The keyword `arguments` hold all the parameters that you pass to a function. `arguments` is an *array like* value.

```
function greet(firstname, lastname, language) {
        if (arguments.length === 0) {
            console.log('Missing parameters!');
            return;
        }
        console.log(arguments);
        console.log('arg 0: ' + arguments[0]);
    }

    greet(); // Does not break
    greet('John'); // Sets 'John' to the first parameter

    // in ES6 I can do:  function greet(firstname, ...other)
    // and 'other' will be an array that contains the rest of the arguments
```

Overloading

Just create a function with different names.

```
function greet(firstname, lastname, language) {
        ...
    }

    function greetEnglish(firstname, lastname) {
        greet(firstname, lastname, 'en');
    }

    greetEnglish('John', 'Doe');
```

Automatic semicolon insertion

Always put semicolons to avoid automatic insertion problems.

Because there is a new line after `return` javascript automatically inserts a semicolon. Use `return {` instead

```
// unreachable code after return statement
```

```
    function getPerson() {
        return
        {
            firstname: 'Tony'
        }
    }

    console.log(getPerson());
```

Immediately Invoked Functions Expressions (IIFEs)

Run the function at the point it is created. Add a `()` after its declaration.

```
// function statement. Have to have a name.
    function greet(name) {
        console.log('Hello ' + name);
    }
    greet('John');

    // using a function expression
    var greetFunc = function(name) {
        console.log('Hello ' + name);
    };
    greetFunc('John');

    // using an Immediately Invoked Function Expression (IIFE)
    var greeting = function(name) {
        console.log('Hello ' + name);
    }('John');
```

IIFE

```
var firstname = 'John';

    //Function statement wrapped in parenthesis so javascript treat is as valid
    // Remember function statement have to have a name.
    (function(name) {

        var greeting = 'Inside IIFE: Hello';
        console.log(greeting + ' ' + name);

    }(firstname)); // IIFE Execute the function on the fly
```

IIFE and Safe code

Frameworks normally use IIFE to have its code close to external code.
An IIFE runs in its own execution context.
A framework would start with a parenthesis and close with another.

```
```

```
(function(global, name) {
        var greeting = 'Hello';
        global.greeting = 'Hello'; //We can access the global object passing it as
a parameter.
        console.log(greeting + ' ' + name);

    }(window, 'John')); // IIFE
```

Closures

Closures allow us to access variables whose execution context has been remove from execution stack. `greet` context is removed after `var sayHi = greet('Hi');`is executed, but `sayHi('Tony');` still have access to the variable `whattosay` that is needed to execute the annonymous function inside `greet` This is because javascript allows executon contexts to closes in its outer variables.

```

function greet(whattosay) {

       return function(name) {
           console.log(whattosay + ' ' + name);
       }
    }

    var sayHi = greet('Hi');
    sayHi('Tony');
```

```

function buildFunctions() {
        var arr = [];
        for (var i = 0; i < 3; i++) {
            let j = i;
            arr.push(
                function() {
                    console.log(i);// Returns always 3, because the last state of
the i val in memory is 3
                    console.log(j);// Returns 0,1,2, beacuase 'let' is scope to the
block, and makes a new variable every time in a different scope.
                }
            )
        }

        return arr;
    }

    var fs = buildFunctions();

    fs[0]();
    fs[1]();
    fs[2]();
```

To make it work in prior ES6 We need to have `i` in a new execution context everytime the loop runs. Use IIFE

```
function buildFunctions2() {

        var arr = [];

        for (var i = 0; i < 3; i++) {
            arr.push(
                (function(j) { // This function will be executed in a differnt
execution context,
                               //  and all of them will be closed in in its closure.
                    return function() { // We push the return of this function. So
when this is call, instead of going up until the loop context, it only goes until
the IIFE where 'j' was stored.
                        console.log(j);
                    }
                }(i))
            )
        }
        return arr;
    }
```

Function Factories

```
function makeGreeting(language) {

        return function(firstname, lastname) {

            if (language === 'en') {
                console.log('Hello ' + firstname + ' ' + lastname);
            }

            if (language === 'es') {
                console.log('Hola ' + firstname + ' ' + lastname);
            }
        }
    }

    var greetEnglish = makeGreeting('en'); // They were created in differnt
execution contexts.
    var greetSpanish = makeGreeting('es');
```

Callback function

A function you give to another function when the other function is finished.

Function

Is a special type of object

It has:

- Name: optional

- Code: invocable using '()'
- bind() set 'this' to what we pass as parameter
- call() like bind but it also execute and can get parameters
- apply() same as call, but parameters have to be an array

```
var person = {
        firstname: 'John',
        lastname: 'Doe',
        getFullName: function() {

            var fullname = this.firstname + ' ' + this.lastname;
            return fullname;

        }
    }

    var logName = function(lang1, lang2) {
        console.log('Logged: ' + this.getFullName());
    }

    var logPersonName = logName.bind(person);
    logPersonName('en');

    logName.call(person, 'en', 'es');
    logName.apply(person, ['en', 'es']);
```

Function borrowing

```
var person2 = {
        firstname: 'Jane',
        lastname: 'Doe'
    }

    console.log(person.getFullName.apply(person2));//or call()
```

Function currying

A copy of a function with preset parameters.

```
function multiply(a, b) {
        return a*b;
    }

    var multipleByTwo = multiply.bind(this, 2);
    console.log(multipleByTwo(4)); // 8

    var multipleByThree = multiply.bind(this, 3);
    console.log(multipleByThree(4)); //12
```

Functional programming

```
function mapForEach(arr, fn) {

        var newArr = [];
        for (var i=0; i < arr.length; i++) {
            newArr.push(
                fn(arr[i])
            )
        };

        return newArr;
    }

    var arr1 = [1,2,3];
    console.log(arr1);

    // Sample 1 add a function
    var arr2 = mapForEach(arr1, function(item) {
       return item * 2;
    });
    console.log(arr2); // [2,4,6];

    // Sample 2 diffent output type
    var arr3 = mapForEach(arr1, function(item) {
       return item > 2;
    });
    console.log(arr3); //[false,false,true]

    // Sample 3 Add function with new parameters
    var checkPastLimit = function(limiter, item) {
        return item > limiter;
    }
    var arr4 = mapForEach(arr1, checkPastLimit.bind(this, 1)); // bind is needed so
we have a function that use only one parameter.
    console.log(arr4); //[false,true,true]

    // Sample 4 Simplify function with new parameters
    //We can also simplified this not to use bind all the time
    var checkPastLimitSimplified = function(limiter) {
        return function(limiter, item) {
            return item > limiter;
        }.bind(this, limiter);
    };

    var arr5 = mapForEach(arr1, checkPastLimitSimplified(1));
    console.log(arr5);
```

Try to use always unmutable objects and return always new objects

Udnerscore.js

Underscore lodash

```
var arr6 = _.map(arr1, function(item) { return item * 3 });
    console.log(arr6);

    var arr7 = _.filter([2,3,4,5,6,7], function(item) { return item % 2 === 0; });
    console.log(arr7);
```

Object Oriented JavaScript Prototypal Inheritance

Inheritance

One object gets access to the properties and methods of another object.

Classical Inheritance

Java, C++, ...

Prototypal Inheritance

JavaScript Simple, flexible, extensible.

Prototype an object that is used (Inherited) by another object.
Prototype chain. The chain of Prototypes that an object has.

```
var person = {
        firstname: 'Default',
        lastname: 'Default',
        getFullName: function() {
            return this.firstname + ' ' + this.lastname;
        }
    }

    var john = {
        firstname: 'John',
        lastname: 'Doe'
    }

    // don't do this EVER! for demo purposes only!!!
    john.__proto__ = person;
    console.log(john.getFullName()); // John Doe. 'this' will be set in the
prototype from the object origianted the call. In this case John.
    console.log(john.firstname); // John
    var jane = {
        firstname: 'Jane'
    }

    jane.__proto__ = person;
    console.log(jane.getFullName()); // Jane Default
```

Reflection

An object can look at itself, listening and changing its properties and methods

```
// don't do this EVER! for demo purposes only!!!
    john.__proto__ = person;

    for (var prop in john) {
        console.log(prop + ': ' + john[prop]); // prop = firstname, lastname and
getFullName
        if (john.hasOwnProperty(prop)) { // if is the object not in the prototype
            console.log(prop + ': ' + john[prop]); // prop = firstname and lastname
        }
    }
```

Extend (Assign) Underscore library that place the properties in the object passed.

```
var jane = {
        address: '111 Main St.',
        getFormalFullName: function() {
            return this.lastname + ', ' + this.firstname;
        }
    }

    var jim = {
        getFirstName: function() {
            return firstname;
        }
    }

    _.extend(john, jane, jim); //John will have the same properties as jim and jane

    console.log(john);

    //Returns
    address: "111 Main St."
    firstname: "John"
    getFirstName: ()
    getFormalFullName: ()
    lastname: "Doe"
```

Building Objects

new

`new` set the this keyword to a new empty object
If nothing is return from that function, instead of returning *undefined* it will return an empty object.

Function constructor. A function that lets me construct an object

```
function Person(firstname, lastname) { // This is a function constructor.

        console.log(this); // Returns an empty object
        this.firstname = firstname;
        this.lastname = lastname;
        console.log('This function is invoked.'); // It is invoked because we
called `new`

    }

    var john = new Person('John', 'Doe');
    console.log(john); //   Person { firstname: "John", lastname: "Doe" }
```

Function constructors and Prototypes

All functions have a prototype. An empty Object. Use only by the `new` operator.

```
Person.prototype.getFullName = function() {
        returning this.firstname + ' ' + this.lastname;
    }

    var john = new Person('John', 'Doe'); // john points to Person.prototype as its
prototype.
    console.log(john);
    //Returns
    firstname: "John"
    lastname: "Doe"
    __proto__: Person
        constructor: Person(firstname, lastname)
        getFullName: ()
```

Why not having functions in the function constructor?
Every copy of the object will get a copy of the function (space in memory). If we have it in the prototype, only one copy of the function will exist in memory for any number of objects created.

Use Capital letters as first letter in any function constructor to identify it easily as a function constructor and reduce possible errors when calling them. Use Linters to help you with the process.

Built in Function Constructor

Create objects that have primitives

```
var a = new Number(3) // a --> Number {[[PrimitiveValue]]: 3}

    var b = new String("Hugo") // b --> String {0: "H", 1: "u", 2: "g", 3: "o",
length: 4, [[PrimitiveValue]]: "Hugo"}
    // inside of the String object there is a primitive.
```

Good to add functions to primitives objects

```
String.prototype.isLengthGreaterThan = function(limit) {
        return this.length > limit;
    }

    console.log("John".isLengthGreaterThan(3)); // true

    Number.prototype.isPositive = function() {
        return this > 0;
    }
    console.log(3.isPositive); // ERROR: 3 is not an object.

    var a = new Number(3)
    a.isPositive() // true
```

They are dangeorous better do not use built in function constructor, unless you really need them. You can look for libraries that help you. Like momentjs for Dates, instead of building your owns function constructor.

Arrays and for..in

In an array the property name are the indexes, that is why we can use it in square brackets. `myArray[0]`

```
var arr = ['John', 'Jane', 'Jim']
    for (var prop in arr) {
        console.log(prop + ': ' + arr[prop]); // 0:John, 1:Jane, 2:Jim
    }
    // If we add
    Array.prototype.myFeature='cool'; // it will return  0:John, 1:Jane, 2:Jim,
myFeature:'cool'
    //Use normal for loop
```

Object.create and Pure prototypal inheritance

```
var person = {
        firstname: 'Default',
        lastname: 'Default',
        greet: function() {
            return 'Hi ' + this.firstname;
        }
    }

    var john = Object.create(person); // Creates an empty object with the prototype
of person
    john.firstname = 'John'; // Adds value
    john.lastname = 'Doe';
    console.log(john);
```

We can add functions on the fly.

Polyfill

Code that adds a feature which the engine *may* lack

```
// Object.create POLYFILL
    if (!Object.create) {
      Object.create = function (o) {
        if (arguments.length > 1) {
          throw new Error('Object.create implementation'
          + ' only accepts the first parameter.');
        }
        function F() {}  // Creates an empty function
        F.prototype = o; // Set the prototype of the function equal to the object
we passed
        return new F();  // Creates a new empty obecjt, runs an empty function and
points the prototype of the new empty object to what we passed in.
      };
    }
```

The object becomes a

ES6 and Classes

class at the end it is just an object and *extends* is syntactic sugar.

typeof and *instanceof*

```
console.log(typeof 3); //number

    console.log(typeof "Hello"); // String
    console.log(typeof {}); //Object

    var d = [];
    console.log(typeof d); // Object.
    console.log(Object.prototype.toString.call(d)); // better! [object Array]
```

```
function Person(name) {
        this.name = name;
    }

    var e = new Person('Jane');
    console.log(typeof e); // object
    console.log(e instanceof Person); // true

    console.log(typeof undefined); // undefined
    console.log(typeof null); // Object. a bug since, like, forever...

    var z = function() { };
    console.log(typeof z); // function
```

Strict Mode

```
function logNewPerson() {
        "use strict";

        var person2;
        persom2 = {};
        console.log(persom2); //Uncaught exception
    }

    var person;
    persom = {};
    console.log(persom);
    logNewPerson();
```

Method Chaining

Calling one method after another, and each method affects the parent object.

Own Libraries architect as jQuery

Greetr

```
// IEFF needs the global object and the jQuery reference
    ;(function(global, $) {

        var Greetr = function(firstName, lastName, language){
            // Create a new Function Constructor which has the same parameters as
```

```
the called
            // one from the client, but we return a new Object. The new object is
created
            // because we call 'new' to  a Function Constructor
            return new Greetr.init(firstName, lastName, language);
        }

        // No access is possible here from outside the lobrarys
        var supportedLangs = ['en', 'es'];

        // Declaring and creating new variables that are not expose outside
        var greetings = {
            en:'Hello',
            es:'Hola'
        };

        var formalGreetings = {
            en:'Greetings',
            es:'Saludos'
        };

        var logMessages = {
            en:'Logged in',
            es:'Sesion iniciada'
        };

        Greetr.prototype = {
            // Exposed functions
            fullName: function() {
                return this.firstName +' '+ this.lastName;
            },

            validate: function(){
                if( supportedLangs.indexOf(this.language)==-1 ){
                    throw "Invalid language"
                }
            },

            greeting: function(){
                return greetings[this.language] + ' ' + this.firstName + '!';
            },

            formalGreeting: function(){
                return formalGreetings[this.language] + ', ' + this.fullName();
            },

            greet: function(formal) {
                var msg;

                // if undefined or null it will be coerced to 'false'
                if (formal) {
                    msg = this.formalGreeting();
                }
                else {
                    msg = this.greeting();
                }
                if (console) {
                    console.log(msg);
                }

                // 'this' refers to the calling object at execution time
                // makes the method chainable
                return this;
            },

            log: function() {
                if (console) {
                    console.log(logMessages[this.language] + ': ' +
this.fullName());
                }
                return this;
            },

            setLang: function(lang) {
                this.language = lang;
                this.validate();
                return this;
            },

            HTMLgreeting: function(selector, formal){
                if(!$){
                    throw 'jQuery not loaded';
                }
                if(!selector){
                    throw 'Missing jQuery selector'
                }

                var msg;
                if (formal) {
                    msg = this.formalGreeting();
                }
                else {
                    msg = this.greeting();
                }
                $(selector).html(msg);
                return this;
            }

        };

        Greetr.init     = function(firstName, lastName, language){
             // Keep 'this' safe to the future calls. We will always have this
object in 'self'
            var self = this;
            // Default values
            self.firstName = firstName || '<Your first name here>';
            self.lastName = lastName || '<Your last name here>';
            self.language = language || "en"

            self.validate()
        }

        // 'Greetr.init.prototype' is the prototype of the init function. Every
```

```
function
        // '.prototype' is where the prototype is. We wnat that our object
prototype is
        // 'Greetr.prototype'  that we will define ourself.
        // That way we point 'Greetr.init.prototype' to 'Greetr.prototype'.
        Greetr.init.prototype = Greetr.prototype;

        //Attach it to the global object to be accessed everywhere and Alias to G$
        global.Greetr = global.G$ = Greetr;


    }(window, jQuery)); // we call teh function with the window(global) object and
the jQuery reference
```

Bonus

Transpile

Convert the syntax of one programming language, to another.