JAVASCRIPT

**THEORY**

**Understanding JS: The weird parts**

|  |  |
| --- | --- |
| **Jargon** | **Definition** |
| Syntax parser | A program that reads your code and determines what it does and if its grammar is valid.  This is a behind the scenes interpreter which translates your code into computer readable instructions |
| Lexical environment | Where something sits physically in the code you write |
| Execution context | A wrapper to help manage the code that is running.  There are lots of lexical environments. Which one is currently running is managed via execution contexts. It can contain things beyond what you’ve written in your code. |
| Name/Value pair | A name which maps to a unique value. The name may be defined more than once, but it can only have one value in any given context. That value may be more name/value pairs (i.e. another object) |
| Object | A collection of name/value pairs |
| Global execution context | Creates the global object (window), the ‘*this’* variable and the outer environment |
| Global | Not inside a function |
| Hoisting | Before your code becomes executed line by line, the JS engine has already set aside memory space for the variables and functions you have created in the entire code.  Functions are remembered in its entirety, whereas for variables the value is ‘not remembered’, and is therefore given the value *undefined* (if run before the value is set) |
| Undefined | A special keyword – it is a value a variable receives during the ‘creation’ phase of the execution context. |
| Single threaded | One command at a time is executed. JS is single threaded |
| Asynchronous | More than one at a time. The JS engine is synchronous! However, different engines such as the rendering engine or HTTP request can run asynchronously together with the JS engine. But the JS engine itself can only run synchronously |
| Synchronous | One line at a time and in order is executed. JS is single threaded and asynchronous in its behaviour |
| Invocation | Calling a function() using parenthesis |
| Variable environment | Where the variables live and how they relate to each other |
| The scope chain | Has reference to outer lexical environments |
| Scope | Where a variable is available in your code. And if it is truly the same variable, or a new copy. |
| Let **(es6)** | Allows the JS engine to use block scoping. Let is similar to var, however it is only available in the block (curly braces) it is defined in |
| Dynamic typing | You don’t tell the JS engine what type of data variables hold, it figures it out while your code is running  Variable.  Variables can hold different types of values because its all figured out during execution |
| Primitive types | A type of data that represents a single value i.e. not an object  6 types: undefined, null, string, number, Boolean, symbol(es6) |
| Operators | A special function that is syntactically(written) different. Generally operators take 2 parameters and return 1 result |
| Operator precedence | Which operator function gets called first, if on same line of code |
| Operator associativity | What order operator functions get called in: left to right or right to left, when functions have the same precedence  *\*look on MDN for operator precedence and associativity table\** |
| Coercion | Converting a value from one type to another  This happens quite often in JS because it is dynamically typed |
| Object literal | Creating an object using curly braces |
| Namespace | A container for variables and functions  Typically to keep variables and functions with the same name separate. JS does not support this, so you can ‘fake’ it using objects. This is mainly used in frameworks and libraries to prevent name conflicts with other scripts:  Var English = { //script 1  Greet: ‘hello’  }  Var Spanish = { //script 2  Greet: ‘Hola’  } |
| First class functions | Everything you can do with other data types, you can do with functions  Assign them to variables, pass them around, create them on the fly etc… |
| Expression | A unit of code that results in a value (any data type)  It doesn’t have to save to a variable  Var a = 3;  1 + 2;  Var b = {greeting: ‘Hi’}; |
| Statement | Just does work, doesn’t return a value. E.g. if statements |
| Mutate | To change something |
| Arguments | The parameters you pass to a function |
| Whitespace | Invisible characters that create literal ‘space’ in your written code. E.g carriage return (enter key on keyboard), tabs, spaces.  JS allows whitespace, which allows a programmer to type a lot of comments and make the code more readable without any implications |
| Closures | Is an inner function that has access to the outer functions variables – scope chain – regardless of whether the outer function has returned.  It has access to its own variables, the outer functions variables and the global variables |
| Callback function | A function you give to another function, to be run when the first function is finished.  So the function you invoke (call), ‘calls back’ by calling the function you gave when it finishes. |
| Inheritance | One object gets access to the properties and methods of another object |
| Reflection | An object can look at itself, listing and changing its properties and methods |
| Function constructor | A function used to construct an object |
| Polyfill | Code that adds a feature which the engine may lack |
| Syntactic sugar | A different way to type something that doesn’t change how it works under the hood |

**Execution context**

2 phases:

1. Creation - Where the global object, ‘*this’* variable, outer environment and hoisting (variable and function set up is created)
2. Code execution - Runs your code line by line

When you create variables and functions in a global environment, those variables and functions get attached to the global (window) object.

**The scope chain:**

Function a() {

Var a = ‘hi’;

Function b() {  
 var b = ‘hey’;

}

}

b() sits inside a() which sits inside the global execution context. These functions have access to their own individual outer environments.

**Let:**

Var a = 3;

Var b = 4;

If(a<b) {

Let c = true;

}

Console.log(c); //displays error unless placed within the code block it is defined in.

**Operators:**

|  |  |
| --- | --- |
| Arithmetic | + - \* / % ++ -- |
| Assignment | = += -= \*= /= %/ |
| String | + += |
| Comparison | == === != !== > < >= <= |
| Logical | && || ! |
| Ternary (?) | *Variablename* = (*condition*)?*value1*:*value2* |
| Typeof | Typeof *value* //returns data type |
| Delete | Delete *objectName.propertyName* |
| in | Returns true if specified property is in specified object, otherwise false |
| Instanceof | Operator returns true of the specified object is an instance of the specified object  Var cars = [‘Saab’,’Volvo’,’BMW’];  Cars instanceof Array; //true  Cars instanceof Object; //true  Cars instanceof String; //false |

**Booleans and existence:**

Var a;

If(a) { //if true

Console.log(‘something’);

} //nothing returns as an *undefined* variable is a falsey value

//Null, 0, “”, undefined are also all falsey values

As 0 may be a valid number you are looking for, for your code to execute you could amend the if statement to read:

If(a || a === 0) { //if true OR a is equal to 0

Console.log(‘something’);

}

**Function default values:**

Function greet (name) {  
 name = name || ‘sir’; //set default value

Console.log(‘hello ‘+name);

}

Greet();

As no name was specified as an argument on invocation, ‘sir’ will be displayed. If this default value wasn’t set, the console would read: Hello *undefined*

**Frameworks and default values:**

If in HTML you have a few <script> tags, it is treated like one long JS file that is run in order. If the same variable is declared in 2 different libraries or frameworks, then the latter one will overwrite it. This can be corrected using the || operator:

Window.varName = window.varName || “newName";

**JSON (JavaScript Object Notation):**

JSON is inspired by object literal context, however they are different slightly. JSON properties have to be wrapped in quotes. You can convert JSON to objects and vice versa using 2 built in functions:

Var objectLiteral = {

Name: ‘Mary’,

isAProgrammer: true

}

Console.log(JSON.stringify(objectLiteral)); //converts to JSON string

Var jsonValue = JSON.parse(‘{“name”:”Mary”,”isAProgrammer”:true}’);

Console.log(jsonValue) //converted back to JS object

**Functions:**

Functions are a special type of object. Think of your code in a function as just one of the properties of that object (function). Think of functions as more than just a block of code!

Function statement:

Function greet() {

Console.log(‘hi’);

}

//doesn’t return a value until the function is executed

Function expression:

Var anonymousGreet = function() {  
 console.log(‘hi’);

}

//The var name ‘anonymousGreet’ points to where the function lives. To invoke *anonymousGreet();*

You can pass a function in to a function:

Function log(a) { //function statement

a();

}

Log(function() { //invoke the function statement and pass a function  
 console.log(‘hi’);

});

**By value vs By reference:**

By value (primitives only) - create copies. E.g.

Var a = 3; var b; b = a; a = 2;

Console.log(a); console.log(b); //2, 3

By reference (objects) - points at the same location, does not make a copy. E.g. var c = {greeting: ‘hi’}; var d; d = c; c.greeting = ‘hello’; console.log(c); console.log(d); //’hello’ ‘hello’

***this* keyword:**

var c = {

name: ‘the c object’,

log: function() {  
 this.name = ‘update c object’;

console.log(this); //this points to the c object

var setname = function(newname) {  
 this.name = newname;

}

Setname(‘updated again’);

Console.log(this); //this points to the global object, not the c object!

}

}

c.log();

The fact that the inner function *this* keyword points to the global object is seen as a bug by many programmers. (you can verify its location by checking the window object in the console).

To get around this:

Log:function() {  
 var self = this; //self now points to c object. You created a reference

Self.name = ‘updated c object’;

Console.log(self);

Var setname = function(newname) {

Self.name = newname; //self not defined so goes up the scope chain and finds reference to the *this* variable which is pointing to the c object

}

Setname(‘updated again’);

Console.log(self);

}

This now works as the ‘self’ variable is not declared in the inner function, and therefore goes up the scope chain to find it, in which it is found in the log method, which as the var ‘self’ in and creates a reference of the *this* variable, which points to the c object.

**Arrays:**

Arrays can hold any data types and as JS is dynamically typed, they can also hold different types:

Var arr = [1, false, {name:’Ali’,lastname:’Issaee’}, function(name){ var greeting = ‘hello’; console.log(greeting + name); }, “Hello”];

Console.log(arr); //to view array in console

Arr[4](arr[3].name); //run the function in the array and set the name argument as the name property from the object in the array

**Arguments keyword:**

Function greet(name, lastname) {  
 if(arguments.length === 0) {  
 console.log(‘missing parameters’);

Return;

}

Console.log(firstname);

Console.log(arguments[1]); //same as lastname

}

Greet(‘Ali’,’Issaee’); // prints both arguments to the console

Greet(); //’missing parameters’

Spread is new ‘parameter’ in ES6 and it is used via the … syntax:

Function greet(firstname, lastname, …anyname) {

Console.log(anyname);

}

Greet(‘Ali’,’Issaee’,’Hull’,’UK’); //’Hull’,’UK’

*Returns any other arguments which weren’t explicitly defined when creating the function*

**Immediately Invoked Function Expressions (IIFE):**

You can invoke function statements and function expressions.

(function(name) {

Console.log(‘hello ‘+name);

})(‘Ali’); //you can invoke after the function of before the closing parenthesis

Because this function statement is in parenthesis (grouping operator), it now makes it an expression, allowing you to create a function on the fly.

**Closures:**

Every time a function is invoked, a new execution context is formed and every function which is called inside the function, will have access to that execution context:

(below is an example of a function factory – more info on this needed)

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’); //new execution context - 1

Var greetSpanish = makeGreeting(‘es’); //another new execution context - 2

greetEnglish(‘Ali’,’Issaee’); //1

greetSpanish(‘Ali’,’Issaee’); //1

1 points to 1 execution context (in memory, as function has already returned, but thanks to closure, inner function still has access to the language variable). And 2 points to 2.

**setTimeOut:**

built in function in which you can set a timer when you want the function to run:

function sayHiLater() {  
 var greeting = ‘hi’;

setTimeOut(function() { //setTimeOut function (closure). Note that the function is the parameter to the setTimeOut function

console.log(greeting);

}, 3000); //time in milliseconds

}

sayHiLater(); //invoke

**Callback function:**

Function tellMeWhenDone(callback) { //name of parameter can be anything  
 var a = 1000;

Var b = 2000;

Callback(); //runs the code I passed as a parameter

}

tellMeWhenDone(function() { //invoke function and pass callback function  
 console.log(‘I am done’);

});

**Call(), apply() & bind():**

All functions have access to these methods. They all refer to where the *this* variable points to.

Bind:

Var person = {  
 firstname: ‘Jon’,

Lastname: ‘Doe’,

getFullName: function() {

var fullname = this.firstname + this.lastname;

return fullname;

}

}

Var logName = function() {  
 console.log(‘logged: ‘+this.getFullName());

}.bind(person); //you can bind here…

Var logPersonName = logName.bind(person); //…or here. Person refers to the object you want the *this* variable to point to

logPersonName(); //invoke

Call:

Call() just invokes a function but lets you pass where *this* points to:

logName.call(person); //you can also pass your parameters after the obj name

Apply:

Apply() does exactly the same as call(), except function parameters are passed as an array:

logName(person, [‘param1’,’param2’]);

When would you use these methods:

1. Function borrowing
2. Function currying
3. Var person2 = {

Firstname: ‘jane’,

Lastname: ‘Doe’

}

Console.log(person.getFullName.apply(person2)); //borrowed method from person obj

1. With the use of bind() – as this creates a copy of the function

Function multiply(a,b) {

Return a \* b;

}

Var multiplyByTwo = multiply.bind(this,2); //passed the *a* parameter

Console.log(multiplyByTwo(8)); // 16. b parameter passed here

Function currying creates a copy of a function but with some pre-set parameters. This is very useful in mathematical situations.

**Inheritance & prototype:**

Prototypal inheritance as seen in JS is simple, flexible, extensible and easy to understand, when compared to more classical ways in other programming languages.

All objects (including functions) have a prototype property, which is another object with its own properties and methods.

If a property or method you call isn’t in the specified object, JS will go down the prototype chain to find it, until you get to the object object (the last prototype property)

Objects can share the same prototypes as other objects via the prototype chain.

All objects, including functions and arrays, have access to ‘\_\_proto\_\_’, and you can see these on the console. They all have their own individual prototypes for their data type (.e.g array prototype, function prototype), but their prototypes all lead back to the object prototype (object object).

**For…in statement:**

Loops through the properties of an object: for(variable in object) { };

Var person = {  
 firstname: ‘Jon’,

Lastname: ‘Doe’

}

For(var prop in person) {

If(person.hasOwnProperty(prop)) {

Console.log(prop+’: ‘+john[prop]);

}

}

hasOwnProperty() is a method of the object prototype, this checks if the property specified in the parameters are the objects own properties, and not those in the prototypes.

**Function constructor:**

A function used to construct an object:

Function Person(firstname, lastname) {  
 this.firstname = firstname;

This.lastname = lastname;

}

Var jon = new Person(“Jon”,”Doe”);

The ‘*new’* keyword creates an empty object. It is good practice to capitalise the first letter in function constructor in case the ‘*new’* keyword is missed on any instances you create in order to correct this. An error would not display as the function is still valid even without the ‘*new’* keyword.

All functions have a prototype property that starts of its life as an empty object, and unless you’re using the function as a function constructor, it is never used.

It is used only by the ‘*new’* operator. The prototype of a function is NOT the prototype OF the function, it is the prototype of any object created if you are using the function as a function constructor.

Every function you create gets a ‘.prototype’ property – which is essentially an empty object and you can add to it:

Person.prototype.getFullName = function() {  
 return this.firstname + ‘ ‘ + this.lastname;

}

It is considered good practice to put all properties within the function constructor itself, and to create all methods on to the prototype chain. This is due to methods taking up memory space, and if you create a lot of instances this means that each new object you create will have them methods, as opposed to just the constructors prototype having that method once!

**Built-in function constructors:**

Number, String, Date…

Var a = new Number(3); //creates an object with use of the ‘new’ keyword

Number.prototype //in console – to view all possible properties and methods

String.prototype

Date.prototype //new Date(mm/dd/yyyy); => leave blank for todays date

In the console, if you type variableName followed by ‘.’ (or varName.\_\_proto\_\_), the console will show you all available methods and properties.

Just like self-made constructors, you can add properties and methods on to built-in function constructors:

String.prototype.isLengthGreaterThan = function(limit) {

Return this.length > limit;

}

Console.log(‘john’.isLengthGreaterThan(3)); //true

*\*\*Do NOT use built in functions to create objects for primitives, e.g. var a = new Number(3);\*\**

**Pure prototypal inheritance:**

Var person = {

Firstname: ‘Default’,

Lastname: ‘Default’,

Greet: function() {  
 return ‘Hi ‘ + this.firstname;

}

}

Var john = Object.create(person); //Object.create creates an object, the argument (person) points to the object you want to inherit from

John.firstname = ‘John’; //override default properties

John.lastname = ‘Doe’;

Object.create is a newer way (than function constructors) to create objects. If browsers don’t support it, you can use a polyfill, e.g.:

If(!Object.create) {  
 Object.create = function(o) {

If(arguments.length > 1) {  
 throw new Error(‘Object.create implementation’)

}

Function F() {}; //creates an empty function

F.prototype = o; //sets prototype equal to the obj you passed in

Return new F(); //new empty constructor function

}

}

**ES6 and classes:**

ES6(2015) has a different way to set objects called classes (different to classes in other programming languages).

Class Person {

Constructor(firstname, lastname) {

This.firstname = firstname;

This.lastname = lastname;

}

Greet() {

Return ‘Hi ‘+this.firstname;

}

}

Var john = new Person(‘John’,’Doe’);

To set prototype:

Class InformalPerson extends Person { //extends keyword sets prototype

Constructor(firstname,lastname) {

Super(firstname, lastname);

}

Greet() {  
 return ‘yo ‘+this.firstname;

}

}

**Random:**

Unary operator – takes only one operand (e.g. ++ -- !)

Binary operator – takes two operands (+ - / < >)

Ternary operator – takes three operands ((condition) ? (do this) : (else, do this))

**QUESTIONS**

**Part 1**

1. Know all the definitions in the ‘jargon’ table
2. What are the 2 phases of the execution context and explain
3. Explain the scope chain and give a basic example
4. What is the difference between *let* and *var*, and give example
5. Understand the operator table and know which operators belong to which group
6. Create an if statement with a Boolean condition where 0 (falsey value) could be seen as a valid response
7. Create default values for a function
8. Convert an object literal syntax to a JSON string and vice versa
9. Give examples of unary, binary and ternary operators

**Part 2**

1. What is the difference between a function statement and a function expression
2. Pass a basic function in to another basic function
3. What is the difference between ‘by value’ & ‘by reference’
4. How would you get around the inner function of a methods *this* variable pointing to the window object?
5. Use the *arguments* keyword in an example
6. Give an example of how spread works using the … operator
7. Create an IIFE and explain why it is a function expression and not a statement
8. What is a closure? Relate this back to P2Q3
9. Create a setTimeOut() function as a closure, so that the outer function executes 5 seconds after invocation
10. Create a basic call-back function

**Part 3**

1. What do call(), apply() and bind() do, and give an example of each
2. Why would you use these methods above (2 reasons), and explain each one with an example
3. Access the different data types prototype property
4. Use the for…in statement on an object, use the hasOwnProperty() method in the example
5. What is a function constructor and give example – also create instances
6. What does the *new* keyword do?
7. Add methods to a function constructor via .prototype
8. Take a look at data type prototypes and mess around with some of the properties and methods available
9. How can you view all available properties and methods of a variable in the console (2 ways)
10. Use the Object.create() method to create an object and set up a prototype for it
11. Create a polyfill if Object.create() is not available on a specific browser
12. ES6 – create a class and set prototype