**Even more**

**JAVASCRIPT**

Language basics

Javascript is a cross-platform, object oriented computer programming language. This means that javascript can be used on multiple platforms and systems. Object oriented refers to the language which is based on objects.

Today, javascript can be used in different places:

Client-side: run on the browser

Server-side: back-end – able to interact with databases, thanks to node.js.

How to implement Javascript

1. Inline script – javascript in the HTML page using <script> tags
2. External file – using an external .js file

Variables

Primitive data types:

Number: floating point number (always has decimals)

String

Boolean

Undefined: data type of a variable which does not have a value yet

Null: non existent

Primitive means ‘not an object’ AKA simple data types.

\*\*Javascript has dynamic typing – which means you do not have to manually define the data type of a variable (it does this automatically) \*\*

Type coercion

When you have different data types and try to mix them together, javascript tries to make sense of it automatically by converting a data type.

E.g. var name = “John”;

Var age = 26;

Console.log(name+age); => string “John26”

\*\*This would be an error in strongly typed programming languages\*\*

*Comparison operator*

*=== This doesn’t do type coercion* ***(use this mostly!)***

*== this does type coercion*

E.g. if(23 == “23”){

Console.log(‘it worked’);

} =>prints ‘it worked’

If(23 === ”23”) {

Console.log(‘it works’);

} => returns false/blank

Variable mutation

Variable mutation refers to the change of a value in a variable.

E.g. var name = “Ali”;

Name = “John”; => same variable but different value to initial set

Operator precedence

This determines the way in which operators are parsed (interpreted) with respect to each other.

E.g. 3+4\*5 =>23 --- this reads like 3+(4\*5)

This is due to multiplication operator having a higher precedence than the additional operator.

\*\*Look at *operator precedence table* online to see order of precedence\*\*

Brackets/grouping has the highest precedence and should be used if you want to be specific of order.

E.g. 3+4\*4-6; =>17 //5\*4 = 20, – 6=14, + 3 = 17

(3+5)\*4 – 6; =>26 // 3+5 runs first

Different functions

Function statement:

Function name() {

//code

}

Function expression:

Var name = function() {

//code

}}

The difference between expressions and statements are:

Expressions = produce a value. (e.g. 3+4, var x=3)

Statements = performs an action (loops and if statements)

Arrays

*Methods*

|  |  |
| --- | --- |
| Push() | Adds a value to the end of an array |
| Unshift() | Adds a value to the beginning of an array |
| Pop() | Removes last element from an array |
| Shift() | Removes first element from an array |
| indexOf() | Returns index number of element in an array. Returns *-1* if element in parenthesis is not in the array  e.g. var john = [“John”,”Smith”,”Designer”, false, 26];  if(john.indexOf(‘teacher’) === -1) {  console.log(“John is not a teacher!”);  } |

Objects

The most common way to write an object is using *object literal*:

e.g. var ali = {

name: “Ali”,

age: 26

}

The two ways to retrieve object data:

1. Console.log(ali.name); => dot notation
2. Console.log(ali[‘name’]); => similar to array

Data mutation:

1. Ali.name = “Sha”;
2. Ali[‘age’] = 28;

Other way to write an object:

Var ali = new Object(); => creates an empty object

Ali.name = “Ali”;

Ali.age = 26;

Ali[‘job’] = “programmer”;

Objects and methods

Var ali = {

Name: “Ali”,

yearOfBirth: 1991,

calculateAge: function() { =>function expression  
 this.age = 2017 – this.yearOfBirth;

}

}

Ali.calculateAge();

Console.log(ali);

Loops

*Continue; => continue loop*

*Break; => break loop*

For(var i = 0; i<=5; i++) {

If(i === 3){

Break;

}

Console.log(i);

} =>1, 2

For(var i = 0; i<=5; i++) {

If(i === 3) {

Continue;

}

Console.log(i);

} => 1,2,4,5

Behind the scenes

Parser = reads the code line by line and checks if syntax is correct. If it is incorrect then an error is produced.

AST = Abstract syntax tree – if syntax is correct then parser produces a data structure called the abstract syntax tree, which is later converted into machine code.

Machine code – no longer javascript code. Machine code is a set of instructions which can be executed directly by the computers processor.

Execution context and stacks

All Javascript code needs to run in an environment called execution stacks. Think of these like a container which stores variables, and in which a piece of our code is evaluated and executed.

The default = the global execution context (code that is not inside any function).

*Execution stacks example*

var name = “john”;

function first() {

var a = “Hello “;

second();

var x = a+name;

}

Function second(){  
 var b = ‘hi’;

Third();

Var z = b+name;

}  
 function third(){

Var c =”Hey”;

Var z = c+name;

}

First();

Execution context in detail

Execution context objects store – variable object, scope chain and the ‘this’ variable.

Hoisting is the JS interpreters action of moving all variable and function declarations to the top of the current scope. However, only the actual declarations are hoisted.

calculateAge(1991) //=> This can run before the function has been written

function calculateAge(year) {

console.log(2017 – year);

} //=> function declaration

Retirement(1956); //=> This does not work as it is an expression

Var retirement = function(year) {

Console.log(65-(2017 – year));

}

*Retirement(1956); //=> now it will work*

Scoping = each new function creates a scope, this is the space/environment in which the variables it defines are accessible.

Lexical scoping = a function that is lexically within another function gets access to the scope of the outer function. (this does not work backwards).

The *this* keyword

In a regular function call – this *this* keyword points at the global object (window). Even if a regular function call is in a method.

In a method call – the *this* keyword points to the object that is calling the method.

Method borrowing

You can borrow methods from other objects which use the *this* keyword to prevent repeating code. E.g.

Var john = {

yearOfBirth: 1990,

calculateAge:function() {

console.log(2017 – this.yearOfBirth);

}

}

Var mike = {

yearOfBirth = 1964

}

Mike.calculateAge = john.calculateAge;

Mike.calculateAge();

DOM

In plain javascript, to change CSS you can use ‘style’ property followed by the CSS property, before setting the value:

Document.querySelector(‘.dice’).style.display = “none”;

*\*\*querySelector use the same selector syntax as CSS\*\**

Events

Google search MDN event reference for full list of different events.

Use anonymous functions in event listeners if you are not planning on using that function anywhere else in your code.

*Ternary operator (shorthand if else statement)*

activePlayer === 0 ? activePlayer = 1 : activePlayer = 0;

*comparison arg. Do this else do this*

To toggle class in plain JS:

Document.querySelector(‘.player-0-panel’).classList.toggle(‘active’);

\*\**toggle* can be replaced with *add* or *remove* in order to add or remove specific classes permanently\*\*

DRY = Don’t Repeat Yourself

Never repeat the same code. Use a function method if you plan on using the same code more than once.

Create an init() function to refer back to the start position of a code. Init = initialisation.

State variable

State variable – simply tells us the condition of a system. E.g. var gamePlaying = true;

This should then be placed in an if statement:

Var gamePlaying = true;

Var el = document.getElementById(‘el’);

El.addEventListener(‘click’, function() {

If(gamePlaying) { //=> if true

//do code

} //=> if false, nothing happens. Or create other code in else

//set active player to *false* where you want the code to stop. i.e once the game is finished (there is a winner).

Advanced Javascript

Objects & Functions

**Objects**

|  |  |
| --- | --- |
| **Primitives** | **Everything else….** |
| Numbers | Arrays |
| Strings | Functions |
| Booleans | Dates |
| Undefined | Wrappers for numbers, strings and Booleans |
| null | **… is an objects** |

*Object Oriented Programming –* Makes heavy use of objects, properties and methods. And these objects interact with each other to form complex applications.

Constructor and instances

*Constructor (or prototypes)* - are like a blueprint.

|  |  |  |  |
| --- | --- | --- | --- |
| **Constructor** |  | **Instance 1** | **Instance 2** |
| *Person* |  | *Var jane* | *Var mark* |
| Name  yearOfBirth  job  calculateAge |  | Jane  1948  Retired  calculateAge() | Mark  1969  Designer  calculateAge() |

*Inheritance* - is when one object is based on another object. When one object gets access to another objects properties and methods.

Javascript is a prototype based language which means inheritance works by using something called prototypes.

Each and every JS object has a prototype property which makes inheritance possible in JS.

**PROTOTYPE CHAIN**

Inherit

Inherit

Each and every object that we ever create, is an instance of the object constructor which has a bunch of methods in its prototype property.

*Prototype chain:*

* When we try to access a certain method or property in an object, JS will first try and find that method/property in that object
* If it cannot find it, it will look in the objects prototype, which is the prototype property of its parent
* If not found then it continues until there is no more prototype to look at
* Null = no prototype => undefined

*Summary*

* Every JS object has a prototype property, which makes inheritance possible in JS
* The prototype property of an object is where we put methods and properties that we want other objects to inherit
* The constructors prototype property is not the prototype of the constructor itself, it’s the prototype of all instances that are created through it
* When a certain method/property is called, the search starts in the object itself, if it cannot be found, the search moves on to the objects prototype. This continues until the method is found

Function constructor

\*\*Write function constructors with the first letter capitalised\*\*

Var Person = function(name, yearOfBirth, job) {

This.name = name;

This.yearOfBirth = yearOfBirth;

This.job = job;

}

Var john = new Person (‘John’, 1991, ‘teacher’);

*The new operator creates a new empty object and then the function is called which also has a ‘this’ variable – which points to the empty object that was created, as opposed to the global object.*

\*\*Methods can be added to the function instructor e.g. this.calculateAge = function() { console.log(2017 – this.yearOfBirth); }\*\* - HOWEVER – it is better practice to use inheritance if you do not want all objects using the constructor to access a certain method/methods. (*see below)\*\**

Person.prototype.calculateAge = function() {

Console.log(2017 – this.yearOfBirth);

} //add this in the global scope, not in the constructor function

If we put methods in the constructor, then with every instance we create, we would have that many copies of the exact same method, and imagine if these files were very big with 100s of lines of code it would impact on memory etc…

You can also put properties in prototypes although this is not very common.

The prototype chain in the console

1. View object in console by typing the name of the object in the console
2. Under \_proto\_ you can view all methods and properties in the prototype
3. The chain can be viewed by clicking \_proto\_ on all prototypes until you get to the object constructor. *\*All objects are instances of the object constructor\**
4. All these show all methods available to that object

Creating objects via objects.create

\*Not as popular as function constructor, but it is still used a lot\*

This way of creating objects builds an object that inherits directly from the one that we passed in to the first argument, whereas the function constructor, the newly created object inherits from the objects prototype property.

Var personProto = {

calculateAge: function() {

console.log(2017 – this.yearOfBirth);

}

};

Var jane = Object.create(personProto, {

Name: {value: ‘Jane’},

yearOfBirth: {value: 1969},

job: {value: ‘Designer’}

});

**Functions**

* A function is an instance of the object type
* A function behaves like any other object
* We can store functions in variables
* We can pass a function as an argument to another function
* We can return a function from a function

It is better practice to make more functions which do individual calculations, the to create one big function to calculate everything.

Var years = [1991, 1965, 1937, 2005, 1998];

Function arrayCalc(arr, fn) {

Var arrRes = [ ];

For(var i =0; i<arr.length; i++) {

arrRes.push(fn(arr[i]));

}

Return arrRes;

}

Function calculateAge(el) {  
 return 2017 – el;

}

Function isFullAge(el) {

Return el>=18;

}

Function maxHeartRate(el) {

If(el >= 18 && el <= 81) {

Return Math.round(206.9 – (0.67\*el));

}else{

Return -1;

}

}

Var ages = arrayCalc(years, calculateAge);

Var fullAges = arrayCalc(ages, isFullAge);

Var rates = arrayCalc(ages, maxHeartRate);

Console.log(ages); //do the same for fullAges and rates.

Functions returning functions

Function interviewQuestions(job) {

If(job === ‘teacher’) {

Return function(name) {  
 console.log(name+’, what subject do you teach?’);

}

}else if(job === ‘designer’) {  
 return function (name) {

Console.log(‘What is UX design, ‘+name+’?’);

}

}else{

Return function(name){

Console.log(name+’, what do you do?’);

}

}

}

Var teacherQ = interviewQuestions(‘teacher’); //store

teacher(‘Ali’); //call

//OR

interviewQuestions(‘designer’)(‘Mark’);

Immediately invoked function expressions (IIFE)

(function(){

Var score = Math.random() \*10;

Console.log(score>=5);

})(); //arguments can go in these parameters for the function

\*\*The primary reason to use an IIFE is to obtain data privacy, because JS var scope variables to their containing function, any variables declared within the IIFE cannot be accessed by the outside world.\*\*

Closures

An inner function has access to the variables and parameters of its outer function, even after the outer function has returned.

*going back to the interview questions example above:*

function interviewQuestion(job) {

return function(name) {

if(job === ‘teacher’) {

//code

}else if(job === ‘designer’) {  
 //code

}else{

//code

}

}

}

InterviewQuestion(‘teacher’)(‘John’);

\*\**the ‘job’ argument can be used in the inner function even though the function has returned\**\*

Bind, call and apply

Bind, call and apply are methods for functions.

Call() – allows method borrowing – in the parameters you need the name of the object which is borrowing the method and also the arguments for the method.

Object1.methodName.call(object2, ‘argument1’, ‘argument2’);

*Object1 = object we want to borrow from. Object2 = object we want the method to apply to*

Apply() – this method is the same as the call() method except the arguments for the methods are in an array.

Bind()- is similar, except you store the method you are borrowing from in a variable.

Var store = obj1.method.bind(obj2);

Store(‘arg1’,’arg2’); //call method

*// you can also store arguments in the initial variable. E.g. var store = obj1.method.bind(obj2, ‘arg1’); store(‘arg2’);*

These methods can also be used for functions and when passing functions as arguments.

e.g. var nameOfVar = function1(‘arg1’, function2.bind(‘arg1’,’arg2’));

console.log(nameOfVar);