Javascript

Overview

Javascript is a programming languauge designed to run in a web-browser, allowing a webapp app to be written without the need for page refreshes, create cool moving graphics, and many other benefits such as cookies.

Many concepts are similar to Ruby, such as spaces and line breaks being not inportant, however much of the syntax is different;

Since it is designed to run in the browser, and the majority of web users browse the internet with low power devices, JavaScript is and should be designed to use the least power and memory possible.

One note for Javascript is to always close your brackets after opening them. Since there tends to be many brackets in a JavaScript script, it can get very confusing without a linter, so just make sure they are closed.

Conventions

In javascript camelcasing is used for naming of all objects but constructors. Therefore, any object name longer than one word will have the second and onwards words capitalised:

*var theMeaningOfLife = 42;*

*Thermostat.prototype.increaseTemperature = function() {*

*// some code*

*}*

For functions which returns booleans, the word ‘is’ is normally put infront of the name, however questionmarks cannot be used:

*Dog.prototype.isPurebreed = function() {*

*return true;*

*}*

Methods which are designated to be internal, i.e. not on the public interface of an object, are begun with ‘\_’:

*Banana.prototype.\_myPrivateMethod = function() {*

*// some code*

*}*

Constants (properties which are designed not to change) are defined as block capitals:

*Dog.prototype.NUMBER\_OF\_LEGS = 4;*

Line Enders

Semicolons are used to define the end of a line of code. For example:

*var x = 5;*

Semicolons are only obligatory when there are two for more statements on the same line. JavsScript parsers class a line break ‘\n’ as a statement end unless:

* The statement has an unclosed paren, array literal, object literal, or ends in another way
* The line is -- or ++
* It is a for(), while(), do, if(), or else, and there is no brace {

Since the JavaScript parsers regard semicolons as opitional markers that the statement currently being parsed has ended, semicolons can be ommited for line termination if a statement:

* ends with a curly brace
* ends at the end of the progam
* ends where a if a line break appears, the statement is termianted unconditionally

however it is comon to use them optionally more often to make the code clearer. Some consider it a good habit to terminate each statement (telling the computer to do something) with a semicolon.

Avoid semicolons if:

* After closing curly brakcet, unless assignment statements (var x = {};)
* After round bracket of if, for, while, or switch statement

An exception is inside a for loop, semicolons go after the first and second but not after the third.

Variables/Objects

An object is a collection of named-values, similar to a Ruby hash, although since functions are objects, they can be included as values in an object. As almost everything in javascript is a object, variables are included too.

Javascript objects can be created as a:

* Literal, with no constructor
* Instance, using keyword new from defined constructor object
* Constructor, defines the constructor to create similar objects using new

To create a local variable it must be defined with ‘var’ beforehand, otherwise it will be global (bad!). Variables are standard objects, so they can be numbers, strings, arrays, hashes, functions etc:

Number: *var x = 5;* (only one type of number)

String: *var carname = "Volvo XC60";*  (double or single quotes)

Object: *var car = {type:"Fiat", model:"500", color:"white"};*

Array: *var cars = ["Saab", "Volvo", "BMW"];*

Dates:

Booleans:

Maths:

Regular Expressions:

Primitive values are immutable and have no properties of methods:

* string
* number
* boolean
* null
* undefined

For example with the number ‘ x = 3.5’, the variable x can be changed but the number cannot. However, as with Ruby, other objects in javascript are mutable, being created by reference, not value. Therefore:

*var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}*

*var x = person;*

*x.age = 10; // This will change both x.age and person.age*

The scope of a variable/object/function is the location within it was made. Therefore, if a variable was made in the global scope, it will be able to be used anywhere, however if it was created inside a function, its scope would be that function only.

Using “use strict” at the beginning of a script or function will restrict the creation of variable to only those defined at the beginning of the script, and stop the creation of global variables without using window.<variable-name> to define the variable.

Properties

Named values assigned to objects are called properties.

*var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};*

Object properties can be called by name such as:

*person.firstName => ‘John’*

*person{‘firstName’] => ‘John’*

All properties of an object are setup by the constructor function for the object.

Properties can be looped through using ‘for...in’ loops:

*var person = {fname:"John", lname:"Doe", age:25};*

*var txt = ‘’*

*for (x in person) {*

*txt += person[x];*

*}*

*console.log(txt) =*> ‘JohnDoe25’

Properties can be deleted from an object using the ‘delete <object>.<property>’ keyword. Deleted properties cannot be added again and if deleted on a prototype, all objects which inherited from the prototype will also lose the property.

Functions

Functions in javascript must be defined using the ‘function’ keyword, or similar ES6 syntax. Function names are written in camel case (first letter lowercase) with its parentheses including any given arguments directly after (parentheses must be included, even if no parameters are required), then the function code is enclosed in {}. By default, functions return 'undefined', for any value to be returned functions must use the 'return' keyword:

*const myFunction = function(p1, p2) {*

*return p1 \* p2;*

*}*

The function keyword creates a new function object which can be called in the javascript environment, assigned to a variable, and inlcuded in another function.

When calling a function, parentheses must be included, even if no parameters are required, if parentheses are not included, the whole function will be returned as an object instead. It is important to note that when functions are written, they are not called. Therefore, if a variable is created or changed inside a function, it will not be accessible until the function is called and the code executed.

Anonymous functions are simply functions created without a name. These are normally passed into other functions to perform specific tasks, or as callbacks to perform tasks after asynchronous completion of another function:

var a = ["a", "b", "c"];

a.forEach(function(entry) {

console.log(entry);

});

Functions in JavaScript pass primitive arguments by value, meaning the value of the argument object is passed (Pass by value), not the actual object. Therefore, if a variable is changed in the function, the underlying primitive is not changed. However, if the passed argument is an array or object (hash), the value passed is the reference to the object (Pass by reference). Therefore, the object can be mutated inside the function and changes will be affected outside also.

Function Definitions

In javascript there are four different types of function definition, the first is a function declaration.

1. A function declaration is where the function is made of the function keyword followed by the function name, parenthesis, and braces. Declarations create a variable under the current scope, with the identifier equal to the function name. During interpretation, function declarations are hoisted to the top of the current scope, allowing them to be used before they have been defined in the code. Function declarations have a few built in properties, for example, the #name property will return the function name. An example function delcaration is:

function message(text) {

// send message

}

2. A function expression is defined by a function keyword, followed an optional name, list of parameters, and braces. A function expression creates a function object which can be used in different situations, such as being assigned to a variable, being a method on an object, or being used as a callback. An example of a named function expression is:

const messageSender = function messge(text) {

// send message

}

When a function expression is not given a name, it is called an anonymous function, if the function is assgined to a variable the function name is then inferred from the varible name. While anonymous functions are often used for callbacks, it is best practice to use named function when possible to aid in debugging and readability:

const fun = function() {}

3. Shorthand functions definitions were bought into javascript in ES5 and are a quicker way of defining a function expression in an object. Functions are simply defined by name, followed by parameter list, and braces, which will create a named function in the object. For example:

const collection = {

items: [],

add(...items) {

this.items.push(...items);

},

}

Shorthand functions can also use computed method names, allowing for dynamic methods in certain situations:

const addMethod = 'add'

const collection = {

items: [],

[addMethod](...items) {

this.items.push(...items)

}

}

collection[addMethod]('C', 'Java', 'PHP')

4. Arrow functsion are function expressions which do not create their own execution context, but instead take it lexically from its immdiate outer scope, allowing it to use 'this' from its parent instead of creating its own. Without the lexical scoping of arrow functions, traditional function expressions may require 'this' binding to them using the .bind() method. They are anonymous functions by default and do not contain an arguments object. An example arrow function is:

const messageSender = (text) => {

// send message

}

Since arrow functions are lexically scoped, they can be used to get context which other functions would require binding for since they create their own scope, such as in a set timeout:

class Point {

log() {

console.log(this === myPoint); // => true

setTimeout(() => {

console.log(this === myPoint); // => true

}, 1000);

}

}

Unlike function expressions, arrow functions do not change their execution context depending on invocation. Therefore, care has to be taken when using arrow functions to define methods, since if defined on a prototype, the execution context of the arrow function will be the global scope, not the desired object:

function Period (hours, minutes) {

this.hours = hours;

this.minutes = minutes;

}

Period.prototype.format = () => {

console.log(this === window); // => true

return this.hours + ' hours and ' + this.minutes + ' minutes';

};

const walkPeriod = new Period(2, 30);

walkPeriod.format(); // => 'undefined hours and undefined minutes'

Hoisting

During javascript interpretation, variables and function declarations are hoisted to the top of their scope before code execution. Therefore, no matter where they are defined, they can still be called throughout the code.

A simple variable declaration would be:

var a; << variable declaration

a = 100; << variable assignment

or:

var a = 100; << combined declaration and assignment

If a variable is called before it has been assigned in the code it will return the value undefined. Whereas if the variable was never assigned in the code and it is used, the javascript engine will throw a ReferenceError.

Since variables delcarations are hoisted to the top of their scope, if a variable is not declared but is still assigned it will be added to the global scope. Strict mode in javascript will stop this functionality, by throwing ReferenceErrors for undeclared variables.

Function declarations can be used throughout the code no matter where they are defined, however it is important hoisting precedence:

* Variable assignment over function declaration
* Function declaration over variable declaration

For example:

var double = 22;

function double(num) {

return (num\*2);

}

console.log(typeof double); // Output: number

and

var double;

function double(num) {

return (num\*2);

}

console.log(typeof double); // Output: function

Function Invocation & this

Invocation of a function is how it is called in the program. There are four different types of invocation in javascript, each defining the execution context in a different way:

* function invocation - alert('Hello World!')
* method invocation - console.log('Hello World!')
* constructor invocation - new RegExp('\\d') or new myObject.myFunction()
* indirect invocation - alert.call(undefined, 'Hello World!')

The 'this' keyword in javascript referes to the current execution context of a function, and therefore varies depending on the function invocation:

* global context (function invocation) - 'this' referers to the global object
* inside a function declaration (function invocation) - If the code is not in strict mode, 'this' in a function will relate to the global object, if strict mode is set it will be set to undefined unless the function is called
* inside a method (method invocation) - 'this' referes to the object that owns the method
* in a constructor function (constructor invocation) - 'this' referes to the newly created object instead of the constructor function itself
* in an event - 'this' refers to element which received the event
* inside .call() or .apply() (indirect invocation) - 'this' is the first argument of call or apply
* in an arrow function - 'this' is bound lexically, and takes the execution context from the outer function in which it is defined

Note that the context of a function depends only on its invocation type, and therefore if a function invocation is called from within a method invoction, this will be set to the global object. The .bind() function creates a new function which invocation will have the context as the first argument passed to .bind(). The function can then be executed in any context while maintaining the defined context, for example in a set timeout callback. To bind 'this' to a function simply call: .bind(this), e.g.:

setTimeout(testFunction.bind(this), 1000)

or in an event callback where 'this' relates to the element which the event was triggered on:

document.querySelector('button').addEventListener('click', logger.updateCount.bind(logger));

Methods

Methods are functions which belong to an object and can therefore be called directly from the object. For example, strings have default methods such as the method length which returns length of the string as a number. These methods can be called on the variable to return a value.

*var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";*

*var sln = txt.length;*

*sln => 26*

Methods for an object are generally defined in the constructor function or added later via prototypes. In the example below the .hello method is created with a :

*function Greeter(name){*

*this.hello = function(name){ console.log('Hello ,' + name + '!'); }*

*}*

*var gre = new Greeter;*

*gre.hello('Chris'); => ‘Hello Chris!’*

Constructor Functions

In javascript, all functions can be used like a blueprint to create similar objects with similar behaviour using the 'new' keyword (similar to Ruby classes). By convention if a function is designed to be duplicated in this way, it is called a constructor function and its name will be capitalised. For example, the below constructor function creates a car function object called ‘xc90’ which has the same behaviour as the Car constructor.

*const Car = function(car\_type) {*

*// some car related code*

*}*

*var xc90 = new Car(volvo);*

Javascript has several built in constructors:

|  |  |  |
| --- | --- | --- |
| Object(); | String(); | Number(); |
| Boolean(); | Array(); | RegExp(); |
| Function(); | Date(); |  |

Additional properties and functions can be added to exisiting objects easily, however these new attributes will be specific to the object it is added to:

*james.nationality = ‘English’;* 🡸 add property

*james.name = function() {* 🡸 add function

*return this.firstName + “ ” + this.lastName;*

*};*

However, properties and functions cannot be added a constructor and passed to its children in the same way. Either the attribute must be added in the constructor method or added later via a prototype.

New

The new operator creates instances of user defined objects (constructors), to do this the new keyword does:

* Creates blank object
* Sets the constructor of the new object to that new is referenced on
* Passes new object as this context
* Returns this if function doesn’t return its own object

Prototypes

The javascript constructor’s prototype can be used to add new methods and values to a constructor instance without adding directly to a constructor method. They do this via the instance’s private [[Prototype]] property.

function Dog(name) {

this.name = name

}

*Dog.prototype.bark = function() {*

*console.log(this.name + ' says Woof!')*

*};*

*fido = new Dog('fido');*

*fido.bark();*

Prototypes are used instead of adding properties and methods directly to constructor methods when memory and performance is of concern. Since the prototype is a single object, it can be easily manipulated allowing for quick changes to programs.

Using a constructor method to create new objects creates a whole new instance of each object, each with their own properties and anonymous functions, which when only creating a couple objects, is fine. However, if 10s to 100s of objects are created, each object having their own copy of a method, will use up a lot of memory. So, there is no need to use prototypes for singleton objects, such as controllers that interact with a page a delegate work.

Prototypes allow a method to be defined once, and have each instance build from it. As each instance references the prototype, it only uses one copy of the method saving memory. Using prototypes how does make it impossible to create private methods of variables in the constructor.

Prototypes do have the disadvantage of not being able to access private functions and variables defined in the constructor function, so functions defined in the construtor are called ‘privileged’ since they have access.

Classes

Classes are templates for creating javascript objects which were added in ES6, similar to constructor functions but more advanced. A class always has a constructor function, which initalizes the instance objects upon creation. The constructor can only be called once and can use the super keyword to call the constructor of the super class. Similar to functions, classes can be declared and expressed:

A class declaration is hoisted and can therefore accessed throughout the code no matter where it has been defined:

class Rectangle {

constructor(parameters) {

//assign parameters

}

}

Class expressions can be named, un-named, and assgined to variables and must be defined before use:

const Rectangle = class {

constructor(parameters) {

//assign parameters

}

}

The 'static' keyword defines static methods for the class which can be called without instantiating the class but cannot be called through an instance. Static methods are often used for utility methods on a class.

Static class data properties must be defined outside of the ClassBody declaration, e.g.

Rectangle.staticWidth = 20;

Rectangle.prototype.prototypeWidth = 25;

Another experimental way to add data to classes with including it all in the constructor is using public class fields:

class MyClass() {

// properties - don't depend on the constructor

prop1 = 1

prop2

constructor(someArg) {

this.prop4 = someArg;

}

method1 = () => {}

}

Class Extends

The extends keyword is used in class definitions to create a child class from an already defined class. Instances of child classes have access to the methods defined in their parents (super), so if they are called with a method they do not explictly own, the super class will be checked also. Super can also be used to define properties via constructors:

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} makes a noise.`);

}

}

class Dog extends Animal {

constructor(name) {

super(name);

}

woof() {

super.speak();

console.log(`${this.name} barks.`);

}

}

let d = new Dog('Mitzie');

d.speak(); // Mitzie makes a noise.

d.woof(); // Mitzie makes a noise. // Mitzie barks.

Extends can also be applied to tradition function based constructors, however they cannot extend non-constructible objects.

Comments

Comments are create using the ‘// <comment>’ syntax. Similar use to Ruby’s #.

Accessors

Getter: function to return value from object:

*var person = {*

*language : "en",*

*get lang() {*

*return this.language;*

*}*

*};*

*document.getElementById("demo").innerHTML = person.lang;*

Setter: function to set a value in an object:

*var person = {*

*language : "en",*

*set lang(lang) {*

*this.language = lang;*

*}*

*};*

*person.lang(“en”);*

Setters and getters are used to provide a simpler syntax than a full function for retriving values as properties. The setter and getter methods also secure better data quality.

Define property methods can be used to manually create getter and setters on object after they have been constructed, using the following syntax:

*Object.defineProperty(<object>, ‘<property-name>’, {*

*get : function () {this.property = val;}*

*});*

Arrays

In JavaScript arrays are collections of objects which can be added to, manipulated, and iterated over performing various functions.

Arrays are defined using the standard notation:

var arr = [1,2,3]

Pushed too using .push(val) and searched using .indexOf(val)

Loops

For loop:

for (var i = 0; i < 11; i++) object.method();

Times loop:

times = function(n, iterator) {

var accum = Array(Math.max(0, n));

for (var i = 0; i < n; i++) accum[i] = iterator.call();

return accum;

};

(Underscore.js) has a built in times loop, so if possible include for easier syntax.

var vs. let vs. const

var vs. let (ES5):

* let => limits the scope of the variable to the block, statement, or expression in which it was used
* var => defines the variable globally or locally to the function in which it was called, regardless of block scope

ES6 introduced const:

* the const variable defines that the variable will not be reassigned to another value in its lifecycle and will error if it does.

For example, defining a var inisde of a forEach loop where a anonomus function is run, var will be the same definition throughout the loop, so if the variable is called again for some reason, it will reference the version last modifed in the loop. However, let will isloate the variable to each loop, allowing it to keep its value without modification from other loops.

Using let inside statements such as IFs can allow varibables defined not to effect the variables with the same name located in the same function.

Generally, to keep as concise as possible, when a variable is being declared, it should initally be a const until it is required that it should be reassigned in its lifetime. Once reassignment is required, the variable should be a let, then if further scope is required outside of the block it is assigned in.

Factories

The factory design pattern is used in javascript to create objects with or add to them specific methods, which all have the same scope. This means mutliple objects can have the same functions which all have access to the same variables which are local to the added functions. The purpose of a factory is only to add methods to an object quickly, it doesn’t perform any actions itself:

*function SecretDiary() {*

*var locked = true,*

*contents = "Hey!";*

*function unlock() {*

*locked = false;*

*}*

*function lock() {*

*locked = true;*

*}*

*function read() {*

*if (locked) { return "Nope!"; }*

*return contents;*

*}*

*return {*

*unlock: unlock,*

*lock: lock,*

*read: read*

*};*

*}*

Objects can then have these functions assigned to them:

*var diary = SecretDiary();*

Debugger

JavaScript has a built in step debugger which can be used by including the ‘debugger’ keyword at the point of code in which you want the program to pause for debug. This can be partically useful to see scope and variables available at troublesome points of code.

Async

When code is synchronous, in order for one action to happen (such as posting a photo), if it requires a prerequistes (downloading the photo), that prerequistie must have been completed before the posting of the photo is performed. Therefore, to run synchronous code, it would be important to make one action happen after its prerequities. While this is good for actions such as displaying photos, if one action (such as loading a video) in a webpage loading was taking a long time, it could hold up the whole script from loading the page, this is called blocking.

Applications have threads, each thread can only complete one task at a time, so if there are multiple tasks to run and only one thread, the tasks will run synchronously. Generally threads are linked to the cores of a processor, and since most modern processors have mutliple cores, multipul threads can be implemented, and multiple tasks completed at the same time, making the execution of the code asynchronous (async).

Async in JavaScript

In general Javascript is single threaded, with one main thread, i.e. it will execute synchonously. However, the way JavaScript executes is different to truly syncronous code like Ruby, instead of waiting for the previous function to finish, if a function (like an AJAX) which can take time to complete is called, it will return instantly allowing other code to continue running, making the code psuedo-async.

Webworkers

Webworks can be used to send off some expensive JavaScript processing off to separate threads, called a worker, making the code truly async. This allows simple code to always be run, while expensive code gets processed in a different thread.

While webworkers are useful , they can’t access the Document Object Model (DOM), so it can’t do anything to directly update the UI. Workers are also synchronous by nature, so if another task relies on them, it will have be be performed after the worker has finished. Therfore, applications have to be carefully designed not to create errors from some main thread actions waiting for results from worker actions.

A worker is an object which is created using a constructor, such as Worker(), which runs a named JavaScript file in another thread and global context to the current window.

Since some browsers don’t support webworkers first it is important to check, providing a webworker version if supported, and a traditional version if not. To do this use:

*if (window.Worker) {*

*// Code using the Web Workers*

*}*

*else {*

*// Code not using Web Workers*

*}*

Once you know that webworkers are supported, its time to create one:

worker = new Worker("worker\_script.js");

The worker is now primed to go with its script in its own gobalcontext. It can then be communcated with using the following commands:

* worker.postMessage() - Starts the worker with a single string argument, the worker can then have an event listener waiting to perform a function if it receives the right string as a message.

*self.addEventListener('message', function(event) {*

*self.postMessage(event.data);*

*}, false);*

* worker.onmessage = <function(event){}>; - on return message from the worker it will run the function, any data returned can be accessed through the ‘event.data’. it is also possible to set an event listner on the worker and wait for the worker to perform a self.postMessage to make an event happen in the main script.

*worker.addEventListener('message', function(event) {*

*document.getElementById('result').textContent = event.data;*

*}, false);*

* variable in the function. Useful for making workers change the DOM.
* worker.terminate; - stops worker working. ‘self.close()’ can also be used from inside the worker script.
* worker = undefined; - resets the worker after termination for use again

More methods - <https://www.html5rocks.com/en/tutorials/workers/basics/>

Workers can be made on the fly by defining the worker with blobURL:

*var worker = new Worker(blobURL);*

*worker.onmessage = function(event) {*

*// event.data == 'msg from worker'*

*};*

*worker.postMessage(); // Start the worker.*

Closures

A closure is a combination of a function bundled together with reference to its surrounding state (lexical environment/scope).

Lexical scoping is where a function/variable has its scope (range of functionality) set to the specific block in which is was defined, allowing for management of variables and enviroment. In JavaScript this means a variable defined outside a function (inside the closures scope), can be accessed from inside another function defined after the variable declaration (callers scope), which is not in the direct scope of the variable. However, this is not true the otherway round, the closure function, cannot access the callers scope.

Lexical is used since lexical scoping uses where the variable is declared within the source code to determine where that variable is available.

To create a standard closure create a function and a sub function inside it, which will limit the sub-functions scope to the function it has been defined in. Include any variable declarations/logic/etc in the main function then return sub-function from the main function. When calling the main function and assigning it to a variable, it will create a closure (function with set scope) assigned to that variable, which can then be called and have access to the variables defined in the main functions scope, despite being called external to it.

*function makeAdder(x) {*

*var addTo = x*

*return function(y) {*

*return addTo + y; // this could also directly call x if required*

*};*

*}*

*var add5 = makeAdder(5);*

*var add10 = makeAdder(10);*

*console.log(add5(2)); // 7*

*console.log(add10(2)); // 12*

Closures can be used to emulate private functions/variables in JavaScript by creating a function with various methods which get returned and some which don’t. Those which aren’t returned are effectively private from the scope of the objects calling the returned methods.

*var counter = (function() {*

*var privateCounter = 0;*

*function changeBy(val) {*

*privateCounter += val;*

*}*

*return {*

*increment: function() {*

*changeBy(1);*

*},*

*decrement: function() {*

*changeBy(-1);*

*},*

*value: function() {*

*return privateCounter;*

*}*

*};*

*})();*

Unlike factories, when the function which creates the closure is called, it runs fully creating a whole new version with new variables and scope. Therefore, different closures created by the same function maintain their independence.

The scope of a closure and any object in JavaScript is:

* Local scope
* Outer functions scope
* Global scope

It is important to realise when creating a closure the outer functions scope and which variables could be affected by assignments in closures. For example if looping through items and adding closures, they will share the same scope outer scope, therefore if any variables are used in the closure from the outer scope, they will all be the same causing potential issues if it is expect for them to be reassigned.

This doesn’t work (item is shared and reassigned between same closure scope each time):

*for (var i = 0; i < helpText.length; i++) {*

*var item = helpText[i];*

*document.getElementById(item.id).onfocus = function() {*

*showHelp(item.help);*

*}*

*}*

This does work (different closure is created each time):

*for (var i = 0; i < helpText.length; i++) {*

*var item = helpText[i];*

*document.getElementById(item.id).onfocus = makeHelpCallback(item.help);*

*}*

Callbacks

Callbacks are a design pattern to used to inject functions into higher-order fuctions and then call them as and when they are required, i.e. not immeditaely as the higher-order function is executed. Higher-order fuctions are designed to accept functions (called callbacks) as arguments and then contain the logic for when the callback is executed.

Since callbacks are closures, they have access to the containing functions scope, therefore making it possible for them to use the local varibles without injection as arguments. It also means a function which has a ‘this’ reference in it, which is then passed as a callback will still relate the ‘this’ to its original scope, not that in which it has been callbacked in.

Callbacks are important for two reasons:

* They allow for asynchronous functions to return immeditaely but still perform a specific function once they are completed. i.e. logic waits for async function to finish then executes callback.
* The design allows of polymorphism of the callback function, since so long as the function responds to the correct calls, it can be injected into the calling function, ie creating a XML request function with a callback, then injecting various different functions as callbacks to do different things to the repsonse data. This is a commonly used design pattern even when there are no asyncronous functions, since it allows for flexibility in design.

Example

Printer callback

*function printItem(item) {*

*console.log(item.result.region)*

*}*

Save callback:

*function saveItem(item) {*

*localStorage.setItem(‘user’, item)*

*}*

Function with callback:

*function getRegionForPostcode(postcode, callback) {*

*var url = "https://api.postcodes.io/postcodes/" + postcode;*

*var xmlHttp = new XMLHttpRequest()*

*xmlHttp.open("GET", url, true)*

*xmlHttp.onreadystatechange = function() {*

*if (xmlHttp.readyState == 4 && xmlHttp.status == 200)*

*callback(JSON.parse(xmlHttp.response))*

*}*

*xmlHttp.send(null)*

*}*

Calling function with different callbacks, each callback is only executed once the reponse is got from the XMLHttpRequest;

*getRegionForPostcode(‘postcode’, printItem)*

*// => console logs region of postcode*

*getRegionForPostcode(‘postcode’, saveItem)*

*// => saves response data to local storage*

IFFE

Immediately Invoked Function Expression (IIFE) is a function design pattern which creates and executes an anonymous function immediately upon loading. Functions are written then run:

(function() {

console.log("hi");

})();

In order for the IIFE to work, the function must first be declared using the function keyword and standard syntax for anonymous function, then expressed used ‘()’ immeditaley after it. For a parser to see this code pattern correctly, and not as a declaration with unrelated parentheses directly after it, the function expression needs to be diambiguated from a declaration by enclosing it in paraentheses. The parser will then expect an expression rather than a declaration.

IIFE is performed so that details of creating the function and the function itself are kept private from the scope in which it was made, saving confusion of having many one time use non-needed function names. Also when used with closures to create specific function, it can stops pollution of functions during creation, for example with the code below giving a different value of index for each IIFE closure:

*var elems = document.getElementsByTagName( 'a' );*

*for ( var i = 0; i < elems.length; i++ ) {*

*(function( lockedInIndex ){*

*elems[ i ].addEventListener( 'click', function(e){*

*e.preventDefault();*

*alert( 'I am link #' + lockedInIndex );*

*}, 'false' );*

*})( i );*

*}*

The difference between IIFEs and self-executing functions, is that self-executing function perform recursion, where as IIFE are simply invoked after declaration.

Modules in JavaScript

A module is a function or object which presents a public interface but also keeps parts of it’s implementation private. Modules can be created using various desgin patterns, the two most common ways are the constructor prototype pattern and the exports pattern in IIFEs.

The constructor prototype pattern uses a IIFE prototype on a constructor method to create private methods within the scope of the prototype, but only return and expose specific functions and properties:

function Module(name) {

this.name = name

}

Module.prototype = (function() {

var i = 0

function myPrivateFunc() {

console.log(i++)

}

function myPublicFunc() {

myPrivateFunc()

console.log(this.name)

}

return {

publicFunc: myPublicFunc

}

})()

var m = new Module(‘Chris’)

m.publicFunc()

// => ‘0’ ‘Chris’

m.myPrivateFunc()

// error

The module exports pattern is a development of IFFE where the functions are created are executed, however during the executing specific functions within them are exported to the public interface. Since functions are now encapsulated, with only specific functions exposed and having private variables and related functions which are not available on the public interface/global, they have become module like.

Module exports execute passing ‘this’ (global window in the browser) as an argument called export, any function then required to be exported is saved to the export object as its own method name, making it available outside the scope of the function definition.

*(function(export){*

*var counter = 0*

*var add = function(){*

*counter++*

*console.log(counter)*

*}*

*export.add = add*

*})(this)*

Since using a function with a closure simply runs the function, variables stored inside will get deleted each time it is run, so modules have the advantage of persistence with their private properties, similar to constructed instances, however their implementation is truly private. It also reduces the need for the use of a constructor for simple functions which require consistent local variables.

Exports can be performed in node.js simiarly, however when immediately executing the function with ‘this’, the ‘this’ in node.js will relate to the exports object instead of the global scope. Therefore, to access exported module functions they must first be required and then called using:

*require(“./add”).add*

Another way to assign to the exports object in node.js modules is to use module.exports. For example:

module.exports = <function>

The above module pattern is only one of the many ways to create a module in JavaScript, another way is creating an IIFE which returns the functions to be exported and assign them to a variable. Then the variable, which doesn’t have to the the global scope, can be used to invoke the functions of the IIFE:

*var counter = (function(){*

*var i = 0;*

*return {*

*get: function(){*

*return i;*

*},*

*set: function( val ){*

*i = val;*

*},*

*increment: function() {*

*return ++i;*

*}*

*};*

*}());*

*counter.get(); 🡺 0*

*counter.set( 3 );*

*counter.increment(); 🡺 4*

Export Function

The above module export uses a module design pattern to add the method to imported module

Promises

The callbacks for the xmlHttpRequest code above creates a listener for load on the object, this can be done similarly for any ajax object:

*var img1 = document.querySelector('.img-1');*

*function loaded() {*

*// woo yey image loaded*

*}*

*if (img1.complete) {*

*loaded();*

*}*

*else {*

*img1.addEventListener('load', loaded);*

*}*

*img1.addEventListener('error', function() {*

*// argh everything's broken*

*});*

However, this doesn’t catch images which errored before the listeners where set up, meaning events aren’t always the best way to handle ajax objects.

Promises are a method in JavaScript for ensuring a action will not execute until certain conditions have been met. This can be useful since some functions in JavaScript are asyncronous, meaning if an AJAX request takes long to repond, an action which relies on it might fail as it wouldn’t be blocked by the AJAX request.

Promise terminology:

* fulfilled - action related to promise succeeded
* rejected - action related to promise failed
* pending - action hasn’t fulfilled or rejected yet
* settled - action has fulfilled or rejected

How to create a promise:

*var promise = new Promise(function(resolve, reject) {*

*// do a thing, possibly async, then…*

*if (/\* everything turned out fine \*/) {*

*resolve("Stuff worked!");*

*}*

*else {*

*reject(Error("It broke"));*

*}*

*});*

A promise constructor takes a callback argument with two parameters, resolve and reject. It then does an async function then performs each function. Promises can then be called passing with a .then() block which takes two callback parameters, one for success and one for fail.

Promises work using ‘then’ functions which only run once the previous function which the ‘then’ was called on has completed. .then fuctions can be chained together with an error catcher at the end, if an error occurs the promise will look down the whole chain of thens for a error catcher.

*get('story.json')*

*.then(function(response){*

*console.log("Success!", response);*

*})*

*.then(get('story2.json'))*

*.then(JSON.parse, function(error) {*

*console.error("Failed!", error);*

*});*

where they are defined.

Maps

A map object is very similar to a plain object in JavaScript, however it iterates its elements in insertion order and returns an array of [key, value] in the for…of loop.

Traditionally objects were used for maps, but recently the Map object was added. A map does not contain any keys by default and its key types can be anything, including objects and primitives. It's size can also be retrived easily from <map>.size property, and its perfomance is better in scenarios involving frequent addition and removal of key pairs.

Create a map using the Map constructor:

const myMap = new Map()

While traditional object setters will work on maps due to them still being objects, the map datastructure will be affected, providing no performance advantages. Therefore, to set properties on maps use the following:

map.set(key, value)

map.has(key) ==> boolean

map.get(key)

map.delete(key)