QA Consulting

JavaScript

TRAINING MATERIALS - COURSE HANDOUT

Contacts

elliot.womack@qa.com

team.qac.all.trainers@qa.com

www.consulting.qa.com

CONTENTS

DISPLAYING OUTPUT

`document.write()` – Writes to the DOM
`window.alert()` or `alert()` – Creates a popup
`console.log()` – Writes to the console

VARIABLES

```
var variableString = "hi"
var variableNumber = 3
var variableObject = { name:"Elliott" }
window.alert(variableObject.name);
```

DATA TYPES

- Numbers
- Strings
- Booleans
- Objects
- Null
- Undefined
- Only one number type – **64bit, double**
  - › It's not very accurate, so always times your data by say 100, do the math, then revert it to decimals, 0.1 + 0.1 doesn't always equal 0.2
- NAN stands for Not a number
  - › Toxic, any math performed with it will also output it.
  - › NAN isn't equal to anything, even itself, NAN = NAN is false.
  - › NAN is not greater or less than NAN.
  - › Even though it literally stands for Not A Number, it's type is actually a number.
  - › Number(value) produces NAN if it has a problem.
  - › `parseInt(value,1)` – Second number is radix, always use it or you can get weird results.
- Null = no value
- Undefined = default value for variables and parameters
  - › The missing value, value

LOOSELY TYPED

1. Any type can be used anywhere.
2. Any variable can be used as a parameter to any function

JS is not *untyped*

KEYWORDS

- JS has a weird keyword policy
  - https://www.w3schools.com/js/js_reserved.asp
- There's a lot of reserved words, but the majority aren't even used.

OPERATORS

```
var x = 5;          // assign the value 5 to x
var y = 2;          // assign the value 2 to y
var z = x + y;      // assign the value 7 to z
var z = x * y;
var z = x / y;
var z = x % y;
var z += y;
var z *= x;
```

`&& || ! typeof instanceof`

The **`typeof`** operator returns a string identifying the type of the value.

It's not great, no matter what type of object It is, it will return "object", including arrays and null objects. Unless it's a string or a function, which will return "string" etc.

STRING OPERATORS

```
txt1 = "John";
txt2 = "Doe";
txt3 = txt1 + " " + txt2; //John Doe
txt1 = "What a very ";
txt1 += "nice day"; //What a very nice day

x = 5 + 5; //10
y = "5" + 5; //55
z = "Hello" + 5; //Hello5
```

FUNCTIONS

- First class objects
- Functions can be passed, returned, and stored like any variable.
- Functions inherit from **Object** and can store name/value pairs.
- Functions can appear anywhere an expression can appear
- What JS calls a function, other languages call lambda
- Since JS is **loosely typed** you don't need to declare parameter types in the signature
- Functions share the same namespace as variables
- Functions can defined inside of functions
- An inner function has access to variables and parameters of functions it's within
  - This is known as static/lexical scoping

```
function myFunction(p1,p2) {
return p1*p2;
}
var x = myFunction(4,3); //12

function foo() {}
var foo = function foo() {};
```

Almost the same thing, the difference will be noted later.

ARGUMENTS

When a function is invoked, in addition to its parameters it gets a special parameter called arguments

This object contains all the arguments from the invocation.

It's similar to an array (but strictly not)

Useful when your function takes a lot of arguments you want to process iteratively.

```
function sum(a,b,c,d,e,f,g){
  var i,
  n = arguments.length,
  total = 0;

  for(i = 0; i < n; i++){
    total += arguments[i];
  }
  return total;
  }
```

SCOPE

- JavaScript `{ Block }` does not have scope like most languages.
- The only scope is **function scope.**
- If a variable is declared anywhere in a function, the entire function has access to it.
- If you declare a variable twice, it'll only be created once.
- JS has implied globals, which means if you try and use a variable without declaring it, it'll create a global one for you.
  - This is really bad.

VARIABLE HOISTING

Hoisting is the default behaviour of JS which moves all declarations to the top of the current scope.

“a var statement declares variables that are scoped to the running execution contexts Variable Environment. var variables are created when their containing lexical environment is instantiated and are initialized to undefined when created”

The important bit is **variables are created when their environment is instantiated.**

What this means is that it scans through the code, looking for variable declarations, then it initializes them as undefined, and then will execute code normally line by line.

To avoid any issues with this, its best practice to declare all your variables at the top of your scope, not always initializing them, but at least declare them at the top to avoid confusion.

This also works with methods,

```
function myFunc() {
// stuff
}
var myFunc = function() {
//stuff
}
```

The second one will give us undefined if we use it before it is initialized, the first example doesn’t matter when we call it.

FUNCTIONS – BEST PRACTICE

- 1. Variables go first
- 2. Then function
  - a. Variables
  - b. Functions
- 3. Run code

```
var x = 10;
function print(input){
    var x = 0;
    function log(){
    }
    console.log(input);
}
print(10);
```

FUNCTIONS – RETURN STATEMENT

Every function returns a value, if you don't explicitly return anything, it'll return "nothing"

As soon as a return statement is hit, it will stop executing the function and returns that specified value.

Return statements are fundamental in moving around data

```
function myFunction() {
    return Math.PI;
}
function myFunction(name) {
    return "Hello " + name;
}
alert(myFunction("Elliott")) //would make an alert of "Hello
Elliott" !
```

OBJECTS

- Collections of unordered name/value pairs
- Names are strings
- Values can be of any type, including other objects.
- Think of every object like a little database, columns and values.
- Values can also be expressions, like usual.
- : seperates name and values
- , seperates pairs
- Object literals can be used anywhere a value can appear.

```
//object literal
var myObject = {name: "Jack", grade:"A", level:3};
myObject.name = "Jeff";
Window.alert(myObject.level); //outputs 3
//dynamic object
var exampleObject = new Object();
exampleObject.name = "Jeff";
exampleObject["name"] = "Bob";
exampleObject.output = new function()…
```

OBJECT MAKER FUNCTIONS

Similar to constructors, a function that takes data and creates an object with that data.

```
function maker(name,grade,level)
{
var it = {};
it.name = name;
it.grade = grade;
it.level = level;
return it;
}
var myObject = make("Jack","A",3);
```

NESTED LITERALS

As discussed earlier, you can have objects within objects.

```
var myObject = {
 name: "Jack",
 grade: 'A',
 format:{
          type:'react',
          width:1920,
          height:1080,
          interlace:false,
          framerate:24
     }
 };
```

ARRAYS

- Arrays inherit from Object
- Indexes are converted to strings, and used as names for retrieving values.
- Efficient for sparse arrays
- Not efficient for everything else.
- No need to provide a length or type when creating an array
- Arrays have a length member.
- Always one integer larger than the highest integer subscript
- `[]` is preferred to new **Array();**

```
var myList = ['oats','peas','beans'];
//adding a new member
myList[myList.length] = 'barley';
```

An array almost looks like this, this is what the “indexes are converted to strings and used as names” means, it takes the 0, converts it to a string, then returns that value, just like you could in a normal object.

```
{
0:"oats",
1:"peas",
2:"beans"
}
```

ARRAYS – METHODS

Arrays have a handful of useful methods pre-defined

- Concat
- Join
- Populate
- Push
- Slice
- Sort
- Splice

```
delete array[#]; to delete elements, but leave a hole.
myArray.splice(1,1); //to completely remove it, no hole
myArray= [‘a’,’b’,’c’,’d’];
delete myArray[1];
//[a,undefined,c,d];
myArray.splice(1,1);
//[a,c,d]
```

SWITCH STATEMENTS

Statements can use strings

Case values can be expressions, don't have to be constants.

Useful instead of having a lot of if/elseifs

Cases fall down, so will execute all the code below that case statement until it hits a break

Default cases are ran if no other case is matched

```
switch(expression){
case ';':
case',':
case '.':
    punctuation();
    break;
default:
    noneOfTheAbove();
}
```

EVENTS

Events are a way of linking HTML to trigger JS code.
Such as;

- 1. Clicking on a button
- 2. Loading a page
- 3. Key presses

```
<html>
   <head>
      <script type="text/javascript">
    function sayHello() {
                          alert("Hello World")
                }
</script>
   </head>
   <body>
      <form>
    <input type="button" onclick="sayHello()" value="Say
Hello" />
      </form>
   </body>
</html>
```

STRING

- No character type in JS
- They are Immutable
  - Cannot be changed
- Double and single quotes both work the same way
- **String (**number) converts numbers into a string
- Strings have a length member, along with a lot of methods
  - charAt
  - concat
  - indexOf
  - lastIndexOf
  - match
  - replace
  - search
  - slice
  - split
  - subString
  - toLowerCase
  - toUpperCase

```
var a = "Jeff"
alert(a.subString(1,2)
```

STRING – ESCAPE CHARACTERS

Escape characters are certain character combinations that translate to something differently than plain text

e.g. \n means “new line”, which instead of literally printing \n, it will almost simulate you pressing an enter key. Such as:

```
“Hello\nWorld”
```

Will print out

```
Hello
World
```

- `\'` – single quote
- `\”` – double quote
- `\\` - backspace
- `\n` – new line
- `\r` – carriage return
- `\t` - tab
- `\b` - backspace
- `\f` – form feed
- `\v` – vertical tab
- `\0` – null character

BOOLEANS

A boolean value is either true or false.

Boolean in JavaScript works slightly differently than most languages.

A lot of values can be treated as “Truthy” and some are “Falsy”

A list of “Falsy” values are:

- 1. False
- 2. Null
- 3. Undefined
- 4. “”
- 5. 0
- 6. NaN

Everything else is truthy.
“0” is truthy
“false” is truthy

CONDITIONALS

- `+` used for addition and concatenation
- `+` can also convert things into a number `+"42" = 42`
- `==` checks equality, but does type coercion
- `===` does equality of value and type
- `&&` can be used to avoid null references
- `||` can be used to fill in default values
- `!` returns the opposite of what something is
- `!!` will convert anything to its boolean representation.

```
If(a){
    return a.member;
}
Else{
    return a;
}
//could be written as
return a && a.member;
-------------------------------------------------------
var last = input || defaultValue;
```

If both operators are numbers, it'll add them.

If both operators are strings, it'll concatenate them.

This is a bit of a problem since it's a loosely typed language, you don't always know the type you're working with.

Type coercion is where JS will try and convert the type to make the conditional statement pass, which isn't always desirable.

`1 == "1"` would return true. `1 === "1"` would return false.

ITERATION

For loops work just like any other language.

```
for(var i = 0; i< 10; i++)
{
    console.log(i);
}
```

While/do while work the same as any other language.

```
do{
    Console.log("hi");
}while(true);
```

Enhanced for loops work slightly differently, since there's no strict types you can't say "For every string inside of this object", A for each loop will inherently loop through every key/value pair (aka, attribute/member) that object has, and any that object inherits from, which gets out of control quite fast.

An extra statement is necessary so that you don't end up processing data you don't care about.

```
for(var name in object){
if(object.hasOwnProperty(name){
    //do stuff
}
}
```

This will now only process the code if the object itself has that data, not any of its super classes.

DOM METHODS

- HTML DOM can be accessed via JS Methods
- All elements are defined as objects
- getElementById is a method
- innerHTML is a property

```
<html>
    <body>
    <p id="demo"></p>
        <script>
            document.getElementById("demo").innerHTML =
"Hello World!";
        </script>
    </body>
</html>
```

This code changes the contents of the element with the id of “demo” to “Hello World!”

MORE DOM METHODS

https://www.w3schools.com/js/js_htmldom_document.asp

JSON

JavaScript Object Notation

JSON is a popular format for numerous data streams, based on javascripts easy-to-read syntax for objects.

Pretty much every programming language will expose some data via a service that provides JSON formatted objects, and then they will piece them back together in their native language.

The general process is like this.
Get request to web url -> Store response in a jsonObject -> Work with jsonObject

Because JavaScript already uses the notation for its objects, it's even easier to work with.

```
var requestURL = 'https://raw.githubusercontent.com/womackx/
JSONDataRepo/master/example.json';
var request = new XMLHttpRequest();
request.open('GET', requestURL);
request.responseType = 'json'
request.send();
request.onload = function() {
var requestData = request.response;
var myH1 = document.createElement('h1');
 myH1.textContent = requestData['squadName'];
 document.getElementsByTagName('head')[0].appendChild(myH1);
}
```

This example will make a request to the url, set the request type as json, send the request, create a function that will run on the event of **onload**, which is when the response is received, set the response to a variable, create a h1 tag, set the contents of that tag to equal the "squadName" variable in the response object, then append the h1 tag to the head tag of the page.

BEST PRACTICES

- Programming is not about personal taste
  - › It's about rigor in expression
  - › Clearness in presentation
  - › Product adaptability/longevity
- Style is more important in JS than most languages
  - › It's very soft, it tries to babysit you a lot.
- Discipline is necessary

SYNTAX

Semicolons are optional

JS puts them in for you if you don't, this is very bad!

ASI – AUTOMATIC SEMICOLON INSERTION

There are 3 rules that JS follows when it uses ASI

- 1. When a script or module is parsed from left to right, if a token is encountered that is not allowed by any production of the grammar.
- 2. When a script or module is parsed from left to right, the end of the input stream of tokens is encountered.
- 3. When a token is encountered that is allowed by some production of the grammar but the production is a restricted production and the token would be the first token of a restricted production, and the restricted token is separated from the previous token by at least one line termination.

Restricted production means continue/break/return/throw

#2 means it'll put one in at the end of the file.

RULE 1

If it encounters a character that doesn't make sense, it'll put a semicolon between them.

```
var a = 12
var b = 13
if(a) { console.log(a) }
console.log(a+b)
```

This is an example of it working well, since we want var b to be separate from 12. It also puts a semicolon between the) and }, since they're not allowed next to each other.

RULE 2

```
var a = 12 <- rule 1a
var b = 13 <- rule 1a
var c = b + a
['menu', 'items', 'listed']
.forEach(function (element){
console.log(element) <- rule 1 b
})
```

In this example, the `[` character IS allowed after **a**, so it won't put a semicolon in-between them, which is bad because **a** isn't an array or an object, so can't be accessed like one, causing an error.

RULE 3

```
function returnObject(){
        if (someTrueThing)
        {
            return
            {
                hi: 'hello'
            }
        }
    }
```

In this case, if there is a restricted production keyword (return) and then a newline statement, it will put a semicolon in, which is bad because now our object literal won't be returned, nothing will be returned.

LINTING

Linting is an external program that scans your code to detect errors/bad practices

- Enforces style rules
- Spots difficult to see errors
- Eliminates implied globals

The bad part of this is that it will **hurt your feelings** as it tells you everything you've written is awful. The good part is that it will get you to a point where your code is very readable and secure.

Multiple IDEs have extensions/plugins that enable linting.

JSHint is a good one that lets you paste your code online.

ESLint has a lot of configuration but is hard to use.

CODING STANDARD

- 1. Break a line after any punctuator
- 2. Don't break a line after a name, string, number or operator
- 3. Avoid tricky expressions using comma operator
- 4. Do not use extra commas in array literals
  - a. Good: `[1,2,3];`
  - b. Bad: `[1,2,3,];`
  - c. This is because different browsers interpret JS in different ways, this is one of them!
  - d. Some browsers count the length of the array by elements, some by commas!
- 5. Opening bracket should be on the same line as your function
- 6. Always use brackets when they're optional
  - a. Bad: `If(a) //code`
  - b. Good: `If(a){ //code }`

STRICT MODE

- Strict mode slaps JS on the wrist every time it tries to *help* you.
- Can I create a variable for you? **No, throw an exception!**
- Can I put a semi colon in for you? **No, throw an exception!**
- Can I try and handle this bad code? **No, throw an exception!**
- Can I help you out when you make any mistake? **NO, TELL ME I'M WRONG**!
- To enable this, simply write **'use strict'**; at the top of your JS script
- If you only want it to apply to certain scopes, like a function, put it at the top of your function.
- Strict will make you a better programmer!

READ ONLY PROPERTIES

We can define properties with some settings

- Enumerable (Loopable)
- Configurable
- Writable (Ability to change it)
- Value
- If you try to write to a non-writable property, you wont get an error but it also wont change the value.
- If you have use strict enabled, you will get an error.

```
var obj = {}

Object.defineProperty(obj, 'readOnly', {
    enumerable: false,
    configurable: false,
    writable: false,
    value: "This var is read only"
    });

    console.log(obj.readOnly)
```

DELETING PROPERTIES

If you try to delete a property, but later access it, it won't delete it
Use strict will force it to be deleted.

```
var obj = {a: 100, b:200};
var myVar = 10;
delete obj.a;
console.log(obj);
```

ADVANCED JS

CLASSICAL VS PROTOTYPAL INHERITANCE

Classical is when objects are an instance of a class, classes can inherit from other classes. Prototypal is class free, objects inherit from other objects, and objects contain links to other objects.

```
var oldObject = {
    firstMethod: function() {...},
    secondMethod: function() {...}
};
//creates an object to inherit from the old object
var newObject = object(oldObject);
newObject.thirdMethod = function() {...};
var myDoppelGanger = object(newObject);
myDoppelGanger.firstMethod();
```

LINKAGE

- Objects can be created with a secret link to another object.
  - If an attempt to access a name **fails**, then the linked object will be used.
- The secret link is not used when storing.
- Object(o) makes a new object with a link to object o
  - This function technically doesn't exist, so you have to create it.
- Any object can inherit from any object

```
function object(o) {
function F() {}
f.prototype = o;
return new F();
}
var myNewObject = object(myOldObject);
//creates an empty object that has a pointer to the old
object
myNewObject.name = "Tom";
myNewObject.level += 1;
```

INVOCATION

- If a function is called with too many arguments, the extra ones are ignored
- If a function is called with too few arguments, the missing arguments will be undefined.
- No type checking on arguments
- Four ways of calling a function.
  - Function form
  - Method form
  - Constructor Form
  - Apply Form
- The first two are the most common

FUNCTION FORM

```
functionObject(arguments)
```

METHOD FORM

```
thisObject.methodName(arguments);
thisObject["methodName"](arguments)
```

CONSTRUCTOR FORM

```
new functionObject(arguments)
```

APPLY FORM

```
functionObject.apply(thisObject,[arguments]);
```

AUGMENTING TYPES

In JS we can augment the built in types, such as String and Array.
Normally these classes are final, but JS lets us change them, which is **really** powerful!

- Object.prototype
- Array.prototype
- Function.prototype
- Number.prototype
- String.prototype
- Boolean.prototype

```
String.prototype.woof = function () {
    return this += " woof";
};
var example = "dog goes ".woof();
// dog goes woof
```

CLOSURE

- An inner function that has access to the outer functions scope chain
- Closure has 3 scope chains
  - It's own scope
  - Outer function scope
  - Global scope
- Allows us to write code that is;
  - Creative
  - Expressive
  - Concise
- Closure is a function inside another function.

Example

```
function showName (firstName, lastName) {
var nameIntro = "Your name is ";
 // this inner function has access to the outer function's
variables, including the parameter
function makeFullName () {
    return nameIntro + firstName + " " + lastName;
}
return makeFullName ();
}
showName ("Michael", "Jackson");
// Your name is Michael Jackson
```

SIDE EFFECTS

1. Closures have access to the outer functions variable even after the outer function returns.

Example

```
function celebrityName (firstName) {
    var nameIntro = "This celebrity is ";
    function lastName (theLastName) {
        return nameIntro + firstName + " " + theLastName;
    }
    return lastName;
}

var mjName = celebrityName ("Michael");
// At this juncture, the celebrityName outer function has
returned.
// The closure (lastName) is called here after the outer
function has returned above
// Yet, the closure still has access to the outer function's
variables and parameter
mjName("Jackson");  //This celebrity is Michael Jackson
```

2. Closures store references to the outer function's variables

Examples

```
function celebrityID () {
    var celebrityID = 999;
    // We are returning an object with some inner functions
    return {
        getID: function ()  {
          return celebrityID;
        },
        setID: function (theNewID)  {
            celebrityID = theNewID;
        }
    }
}
var mjID = celebrityID ();
// At this juncture, the celebrityID outer function has re-
turned.
mjID.getID(); // 999
mjID.setID(567); // Changes the outer function's variable
mjID.getID(); // 567: It returns the updated celebrityId
variable
```

This example shows the ability to create private variables in JS.

HIGHER ORDER FUNCTIONS

JavaScript introduced a handful of higher order functions in ES5

A higher order function is a function that takes a function as a parameter

Their goal is to promote immutability as well as condense common operations.

MAP()

Accepts a single parameter function

Executes that function on every item in the array

Used for transforming the contents of the collection

139 characters -> **70** characters

Almost half the code!

Example

```
var exampleList = [1, 2, 3, 4];
for (var i = 0; i < exampleList.length; i++) {
    exampleList[i] += 2;
}
console.log(exampleList);

//or if we use the map function
var exampleList = [1, 2, 3, 4];
console.log(exampleList.map(a=>a+2));
```

FILTER()

Accepts a single parameter function

Executes that function on every item in the array

Used for removing objects that don't meet the criteria of the function

171 characters -> 77 characters

Almost 100 characters less code!

Example

```
var exampleList = [1, 2, 3, 4];
var tmpList = [];
for (var i = 0; i < exampleList.length; i++) {
if (exampleList[i] > 2)
    tmpList.push(exampleList[i]);
}

console.log(tmpList);

//or if we use the filter function
var exampleList = [1, 2, 3, 4];
console.log(exampleList.filter(a => a > 2));
```

FOREACH()

Accepts a single parameter function

Iterates over the list and executes that function each iteration

Basically like map but doesn't return a new collection, just executes code.

Generally just used for shorthand forloops.

Example

```
var sum = 0, exampleList = [1, 2, 3, 4];
for (var i = 0; i < exampleList.length; i++) {
    sum += exampleList[i];
}
console.log(sum);

//or if we use the foreach function
var sum = 0, exampleList = [1, 2, 3, 4];
exampleList.forEach(a => sum += a);
console.log(sum);
```

ES6

ES6 brought us a bunch more features that are incredibly powerful and concise. Promoting readability and efficiency.

CONSTANTS

Whilst before we had to create ridiculously large amounts of code to made a variable un-editable, now we have the const keyword.

Example

```
const PI = 3.1415
```

LET

Let is a new powerful keyword that lets us create variables that conform to normal scope rules, so no more do you have loops that expose their inner variables.

Example

```
for(let i = 0; i < a.length; i++){
    …
}
console.log(i) //undefined
```

Let is now the standard keyword for creating variables, using var should be exception now.

ARROW FUNCTIONS

Also known as “Fat Arrows”

Concise way of writing functions

Creates anonymous functions

Similar to lambdas in other languages.

Does not require the function or return keywords or the use of curly brackets.

Example

```
function f(data) = {
    return data + 1;
}

//Arrow equivalent
(data) => data + 1
```

Can also have multiple parameters

If you require multiple lines in the method, you then require a return statement.

Examples

```
//Multiple parameters
(data,moreData) => data + moreData

//Curly brackets, for multiple line statements
(data)=> {
data = data + 1;
data = data * 3;
return data;
}
//New style
odd = evens.map(v => v + 1)
//Old style
odds = evens.map(function (v) { return v + 1; });
//New Style
let output = (data) => console.log(data);
//Old Style
let output = function (data) { console.log(data); };
```

DEFAULT PARAMETERS

Parameters that will use a default value if one is not provided

Examples

```
let doStuff = (a, b, c) => a + b + c

console.log(doStuff(1, 2)); //NAN, since c is undefined

let doStuff = (a, b, c = 3) => a + b + c

console.log(doStuff(1, 2)); //6, since c is set to 3

let doStuff = (a = 1, b = 2, c = "Dog") => a + b + c

console.log(doStuff()); //What would it output?
```

STRING INTERPOLATION

A much nicer way of concatenating strings.

Example

```
var dog = { age: 10, name: "bob" }
console.log("Age: " + dog.age + " Name: " + dog.name);

//New fancy way
console.log(`Age: ${dog.age} Name: ${dog.name}`);
```

DESTRUCTURING

The ability to extra data from variables without creating new references to them!

Assignment

```
//New Way
var list = [ 1, 2, 3 ]
var [ a, , b ] = list
[ b, a ] = [ a, b ]

//Old Way
var list = [ 1, 2, 3 ]
var a = list[0], b = list[2]
var tmp = a; a = b; b = tmp
```

Parameters - Example

```
function f([name, val]) {
    console.log(name, val)
}
function g({ name: n, val: v }) {
    console.log(n, v)
}
f(["bar", 42])
g({ name: "foo", val: 7 })
function h({ name, val }) {
    console.log(name, val)
}
h({ name: "bar", val: 42 })
```

Default values - Example

```
var list = [7, 42]
var [a = 1, b = 2, c = 3, d] = list
a === 7
b === 42
c === 3
d === undefined
```

CLASSES

We have classes now!

Very good for people transitioning from classical based languages, as well as nice tidy syntax for us to use instead of prototypal syntax.

However It doesn't do it any differently than if you just did the prototypes yourself, it just does it for you instead.

Example

```
class Dog {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    bark() {
        console.log("woof");
    }
}
let dog = new Dog("bob",10);
```

Inheritance

We can also use inheritance with this! Extends keyword as usual.

Example

```
class Animal {
constructor(name, age) {
this.name = name;
this.age = age;
}
}
class Dog extends Animal {
constructor(name, age, pawSize) {
    super(name, age);
    this.pawSize = pawSize;
}
bark() {
    console.log("woof");
}
}

let c= new Dog("bob",10,20.5);
c.age //10
```

Static members

Members that belong to the class, not the instance of the class.

Example

```
class Dog {
constructor(name, age, pawSize) {
    this.name = name;
    this.age = age;
    this.pawSize = pawSize;
}
static defaultDog() {
    return new Dog("default", 0);
}
}

let c = new Dog.defaultDog();
```

JavaScript

ARRAY.FIND()

No longer need to hack a solution together for a common function

Example

```
let tmpArray = [1,2,3,4];
let c = tmpArray.find(x=>x>3); // 4
let d = tmpArray.findIndex(x=>x>2)//2
```

STRING.REPEAT()

The ability to duplicate a string on demand.

Example

```
"dog".repeat(3)
//dogdogdog
```

STRING SEARCHING

No longer need to use indexOf() to figure it out.

Example

```
"hello".startsWith("ello", 1) //true
"hello".endsWith("hell", 14) //true
"hello".includes("ello") //true
"hello".includes("ello", 1) //true
"hello".includes("ello", 2) //true
```

ASYNCHRONOUS PROGRAMMING

This has nothing to do with multi threading!

JavaScript is single threaded and asynchronous.

Synchronous execution is waiting until something is finished to then move on.

Asynchronous execution is not waiting.

Imagine Synchronous programming as all your code being based off a Clock, each tick is a line of code, one after another.

Asynchronous is like having two clocks, they don't really care about each other and can tick independently.

Example - SYNCHRONOUS

You are in a queue to get a movie ticket. You cannot get one until everybody in front of you gets one, and the same applies to the people queued behind you.

Example - ASYNCHRONOUS

You are in a restaurant with many other people. You order your food. Other people can also order their food, they don't have to wait for your food to be cooked and served to you before they can order. In the kitchen restaurant workers are continuously cooking, serving, and taking orders. People will get their food served as soon as it is cooked.

Callbacks

Callbacks are the basic way of handling Async execution

Callbacks are founded on the functional programming paradigm, particularly Higher Order Functions

A function that has a function as an argument

Passing what you want the async operation to do after it's done.

Example – Making a web request to the website, however we have to wait for the response, if we paused execution until that happened it's just wasted execution time, we could do other things while we wait by using it in an async way!

Example

```
function example(cb) {
webRequest(); //pretend method
cb();
}
function log() {
    console.log("Done");
}
example(log);//Execution
//Opposed to
webRequest();
console.log("Done");
```

Normal functions should just use return types, however async functions will hit the return before the async request is finished, making it useless. Hence the need for callbacks.

Sync Example

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');
console.log(data.toString());
console.log("Program Ended");
```

Async Example

```
var fs = require("fs");
fs.readFile('input.txt', (err, data) => {
if (err) {
  return console.error(err);
}
console.log(data.toString());
});
console.log("Program Ended");
```