JS-advanced for hands-on test automation engineers

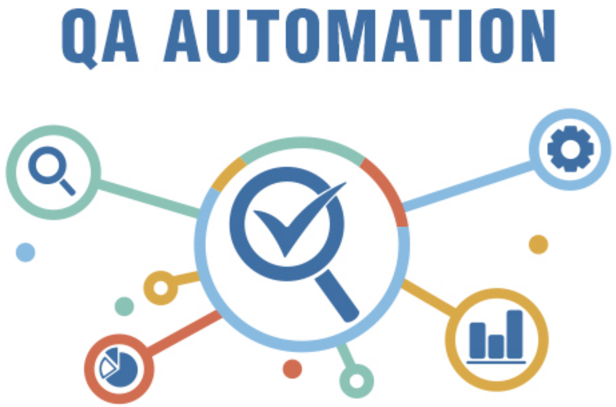
Sydorenko Vladyslav

Overview of course and topics

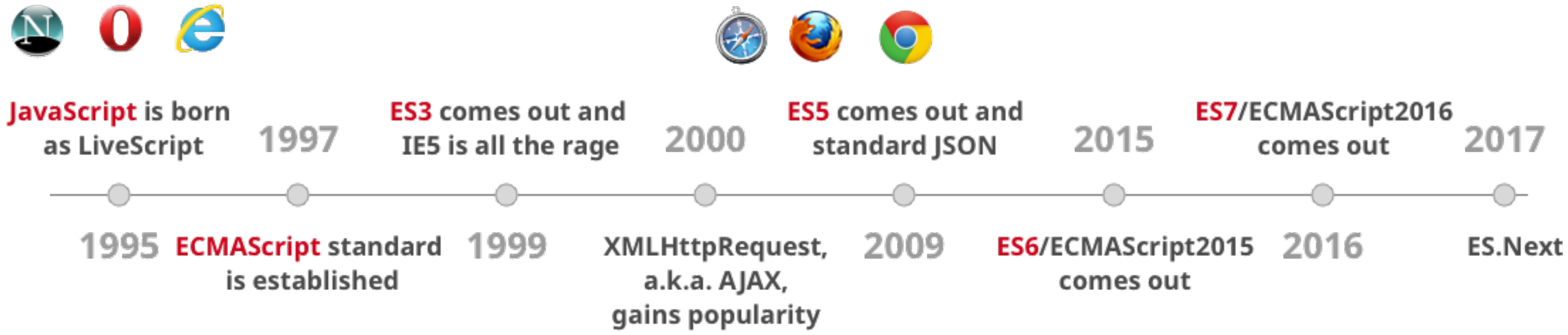
- Basics of JavaScript
- History and purpose of language
- Data types, algorithms
- Functions
    - Context
    - Methods call / apply / bind
    - Arrow style
    - Closure
- OOP
    - JS prototype
    - ECMA6+ style
- Async programming and methods

Auditory

JS dev and Automation QA - to fortify core skills and learn how to use new features of the language



History of JavaScript

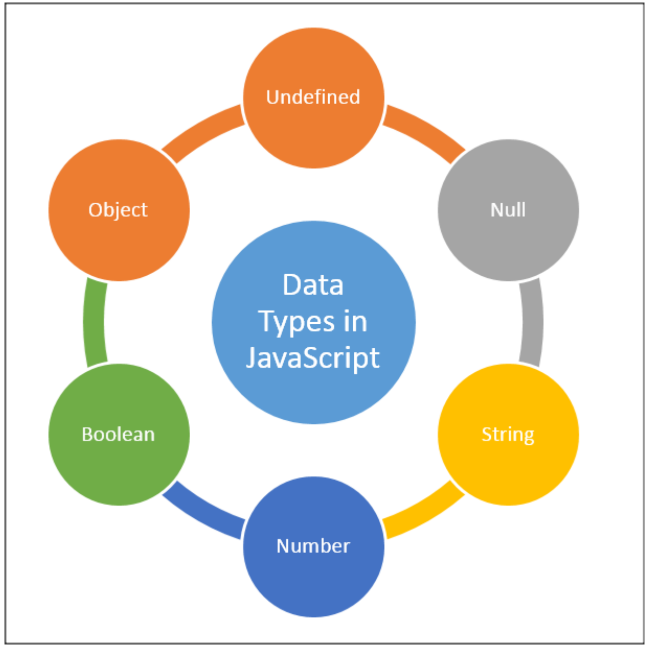

1995
JavaScript is born as LiveScript
1997
ECMAScript standard is established
1999
ES3 comes out and IE5 is all the rage
2000
XMLHttpRequest, a.k.a. AJAX, gains popularity
2009
ES5 comes out and standard JSON
2015
ES6/ECMAScript2015 comes out
2016
ES7/ECMAScript2016 comes out
2017
ES.Next

Data types in JavaScript

THERE ARE 2 MAIN DATA TYPE GROUPS IN JS

Does JS have types?
Some may argue that JS is untyped or that it shouldn’t call its type system types. It doesn’t require you to declare a type when making a variable like in some other strongly typed languages i.e int x = 10 I (and the JS specs) would argue that JS does have types.

JS is both **dynamically typed** and **weakly typed**.



Data types in JavaScript

Dynamically Typed

Dynamically typed languages **infer variable types at runtime**. This means once your code is run the compiler/interpreter will see your variable and its value then decide what type it is. The type is still enforced here, it just decides what the type is.

```
var a = 1 // int

b = 'test' // string
```

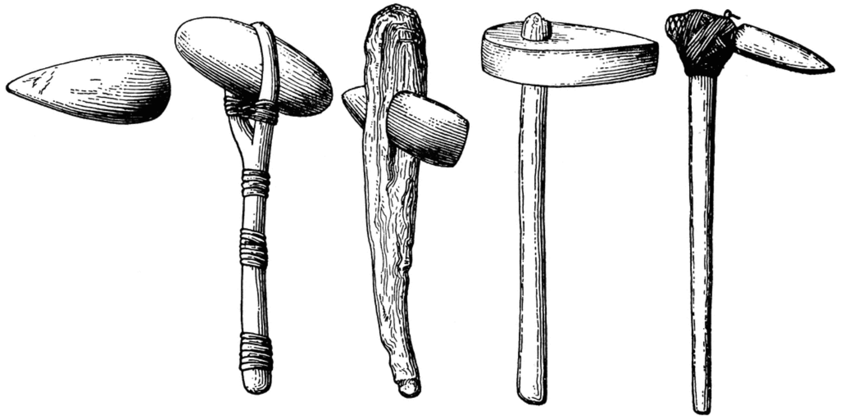
Weakly Typed

Weakly typed languages **allow types to be inferred as another type**. For example, 1 + '2' // '12' In JS it sees you're trying to add a number with a string — an invalid operation — so it coerces your number into a string and results in the string '12'.

Data types in JavaScript

Primitives

- Boolean — true or false
- Null — no value
- Undefined — a declared variable but hasn’t been given a value
- Number — integers, floats, etc
- String — an array of characters i.e words
- Symbol — a unique value that's not equal to any other value



Data types in JavaScript

```
// PRIMITIVE TYPES

// Boolean

true

false


typeof true // 'boolean'


// Null

null

typeof null // 'object' why? answer later


// undefined

undefined

typeof undefined // 'undefined’
```

```
// Number

// 1 1.5 0.5 -1
typeof Infinity // 'number'

typeof 1.5 // 'number'


// String

// 'a' , "b" , `hello`
typeof 'a' // 'string'


// Symbol


typeof Symbol('a') // 'symbol'

Symbol('a') === Symbol('a') // false
```

Data types in JavaScript

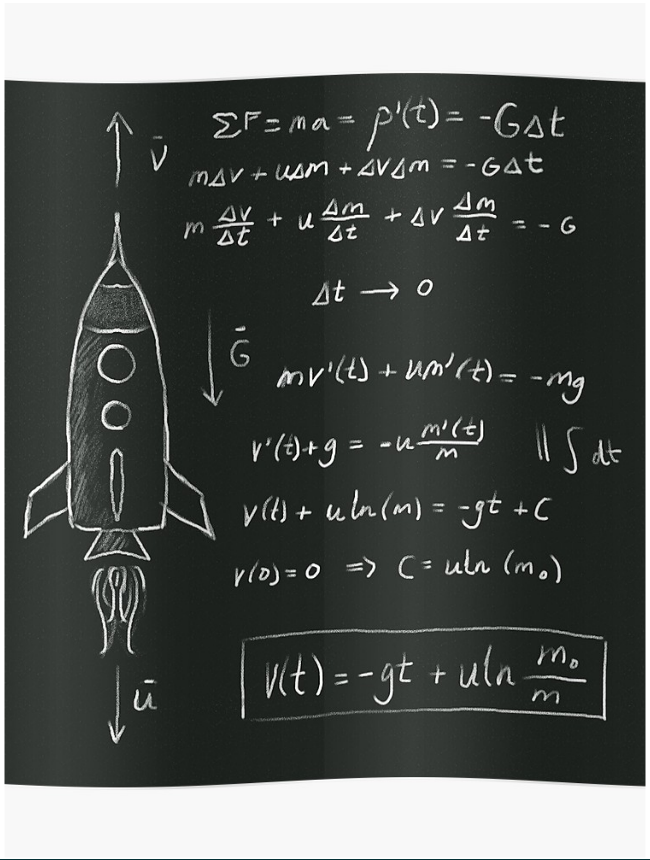
Objects

There are two that are the main ones you'll use for your own structures:

- Object
- Array

There are many other objects too just to list a few:

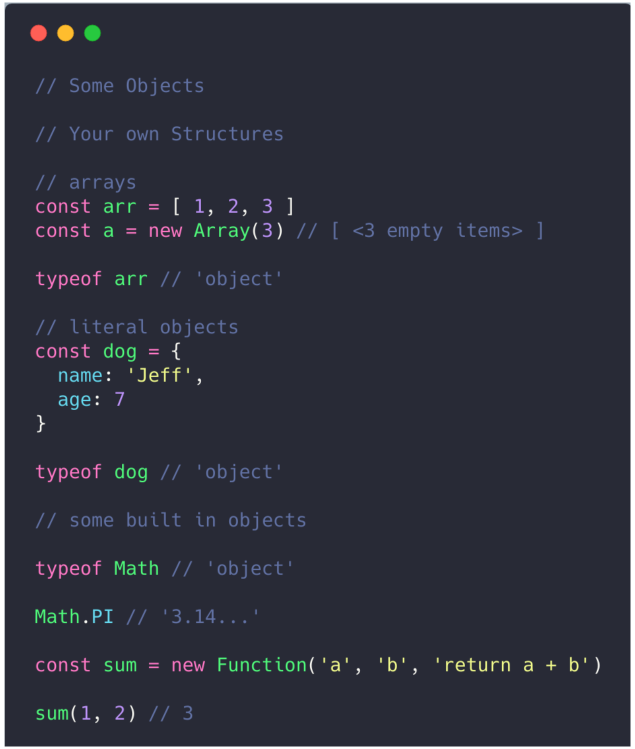
- Function
- Boolean
- Symbol
- Error
- etc



$\sum F = ma = p'(t) = -G\Delta t$
$m\Delta v + u\Delta m + \Delta v\Delta m = -G\Delta t$
$m\frac{\Delta v}{\Delta t} + u\frac{\Delta m}{\Delta t} + \Delta v\frac{\Delta m}{\Delta t} = -G$
$\Delta t \to 0$
$mv'(t) + um'(t) = -mg$
$v'(t) + g = -u\frac{m'(t)}{m}$
$\| \int dt$
$v(t) + u\ln(m) = -gt + C$
$v(0) = 0 \Rightarrow C = u\ln(m_0)$
$v(t) = -gt + u\ln\frac{m_0}{m}$

Data types in JavaScript

```
let objA = {};

objA.a = 'Vlad';

let objB = objA;

objB.a = 'Olga';

console.log(objA.a); // ??
```

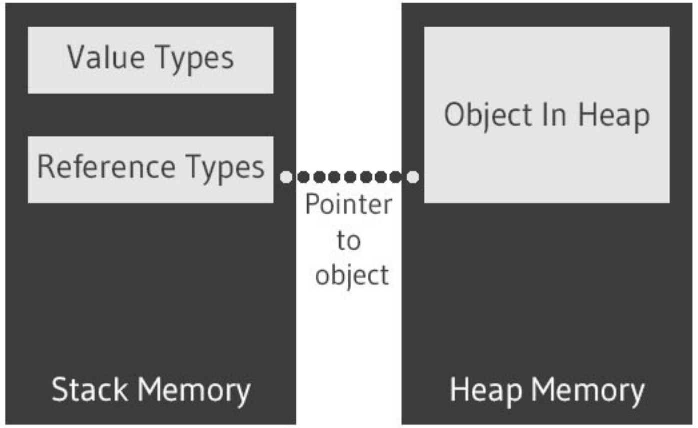


```
// Some Objects

// Your own Structures

// arrays
const arr = [ 1, 2, 3 ]
const a = new Array(3) // [ <3 empty items> ]

typeof arr // 'object'

// literal objects
const dog = {
  name: 'Jeff',
  age: 7
}

typeof dog // 'object'

// some built in objects

typeof Math // 'object'

Math.PI // '3.14...'

const sum = new Function('a', 'b', 'return a + b')

sum(1, 2) // 3
```

Data types in JavaScript

Boxing / unboxing

```
typeof true; //"boolean"
typeof Boolean(true); //"boolean"
typeof new Boolean(true); //"object"
typeof (new Boolean(true)).valueOf(); //"boolean"
typeof "abc"; //"string"
typeof String("abc"); //"string"
typeof new String("abc"); //"object"
typeof (new String("abc")).valueOf(); //"string"
typeof 123; //"number"
typeof Number(123); //"number"
typeof new Number(123); //"object"
typeof (new Number(123)).valueOf(); //"number"
```



Data types in JavaScript

Math link

https://habr.com/ru/post/312880/

Data structures and performance

https://habr.com/ru/post/49052/

Hoisting

WHAT IS VARIABLE HOISTING?

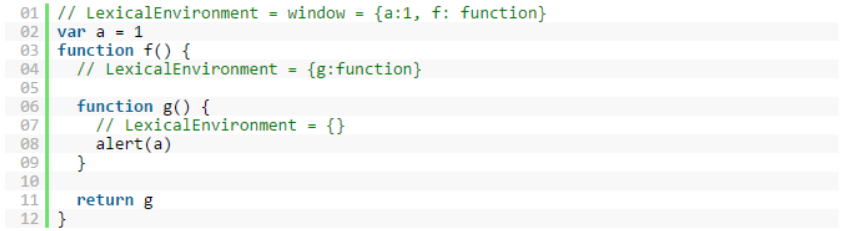
Hoisting

VAR

```
//declaration getting hoisted at the top
var shape;

console.log(shape); // OUTPUT :
undefined

shape = "square"; // OUTPUT :
"square”

console.log(shape);
```

```
function getShape(condition) {
// shape exists here with a value of
undefined
console.log(shape); // OUTPUT :
undefined
if (condition) {
var shape = "square";
 // some other code
 return shape;

} else {
 // shape exists here with a value
of undefined

return false;
}
}
```

Closures



```
// LexicalEnvironment = window = {a:1, f: function}
var a = 1
function f() {
  // LexicalEnvironment = {g:function}

  function g() {
    // LexicalEnvironment = {}
    alert(a)
  }

  return g
}
```

```
function makeCounter() {
let count = 0;
return function() {
return count++; // has access to the outer "count"
};
}
let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Call / Apply / Bind

```
func.call(context, arg1, arg2, ...)

var user = { firstName: ”Vlad", surname: ”Syd", patronym: ”Ivan"
};

function showFullName(firstPart, lastPart) {
alert( this[firstPart] + " " + this[lastPart] );
}

showFullName.call(user, 'firstName', 'surname') // ”Vlad Syd"
showFullName.call(user, 'firstName', 'patronym’) // ”Vlad Ivan”

func.call(context, arg1, arg2); OR func.apply(context, [arg1,
arg2]);

showFullName.call(user, 'firstName', 'surname’);
showFullName.apply(user, ['firstName', 'surname'])
```

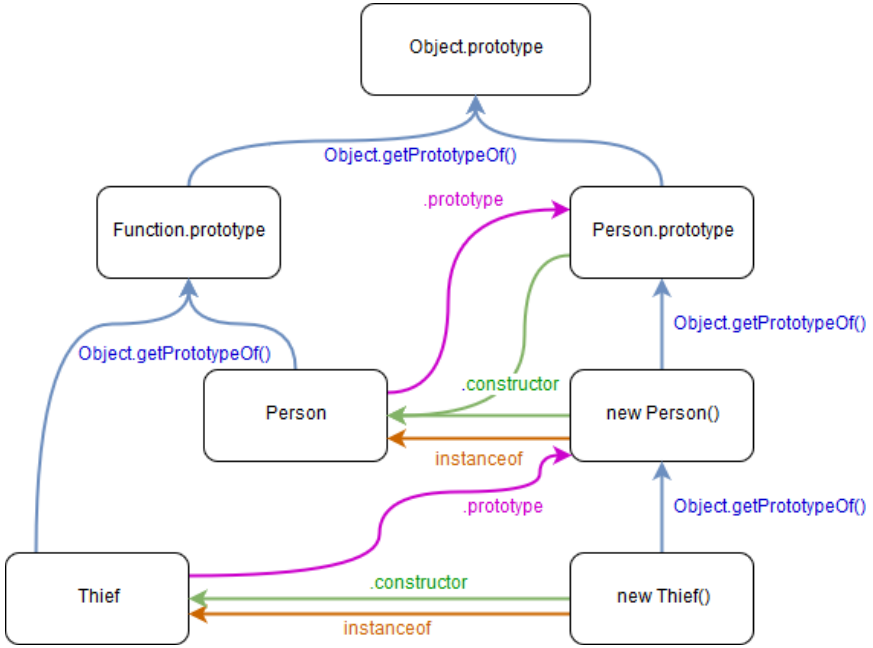
Call / Apply / Bind

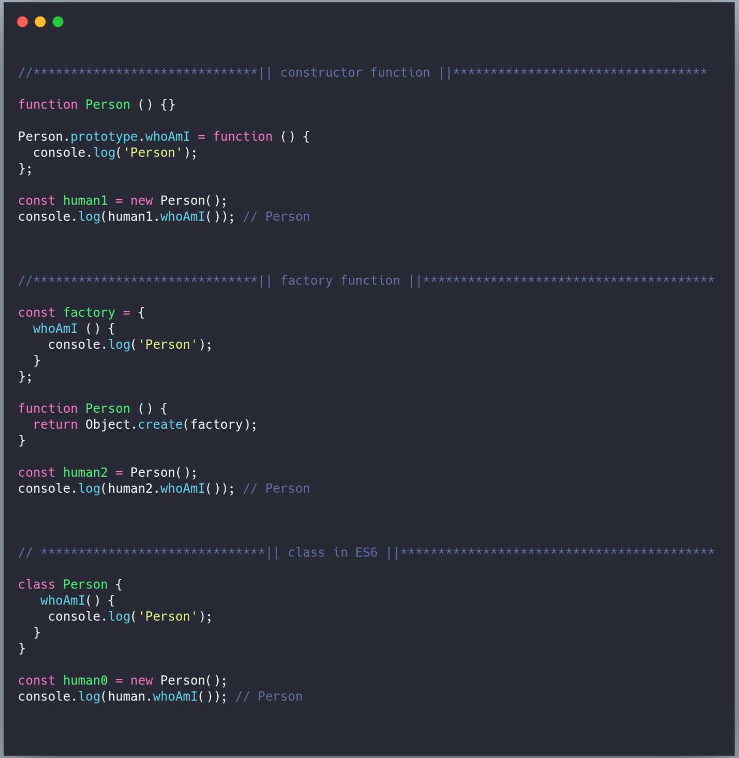
BIND

```
let boundFunc = func.bind(context);

let user = { firstName: "Olga" };
function func() {
       alert(this.firstName);
}
let funcUser = func.bind(user);
funcUser(); // Olga
```

```
function mul(a, b) {
return a * b;
}
let double = mul.bind(null, 2);
alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

OOP in JS





```
//******************************|| constructor function ||**********************************

function Person () {}

Person.prototype.whoAmI = function () {
  console.log('Person');
};

const human1 = new Person();
console.log(human1.whoAmI()); // Person



//******************************|| factory function ||***************************************

const factory = {
  whoAmI () {
    console.log('Person');
  }
};

function Person () {
  return Object.create(factory);
}

const human2 = Person();
console.log(human2.whoAmI()); // Person



// ******************************|| class in ES6 ||******************************************

class Person {
   whoAmI() {
    console.log('Person');
  }
}

const human0 = new Person();
console.log(human.whoAmI()); // Person
```

Arrow functions

Arrow functions – also called “fat arrow” functions, from CoffeeScript (a transcompiled language) — are a more concise syntax for writing function expressions. They utilize a new token, =>, that looks like a fat arrow. Arrow functions are anonymous and change the way this binds in functions.

```
// ES5
var multiplyES5 = function(x, y) { return x * y; };
// ES6
const multiplyES6 = (x, y) => { return x * y };
const multiplyES6 = (x, y) => x * y;

//ES5
var phraseSplitterEs5 = function phraseSplitter(phrase) { return phrase.split(' '); };
//ES6
const phraseSplitterEs6 = phrase => phrase.split(" ");

// ES6
const prices = smartPhones.map(smartPhone => smartPhone.price);
const divisibleByThrreeES6 = array.filter(v => v % 3 === 0);
```

Arrow functions

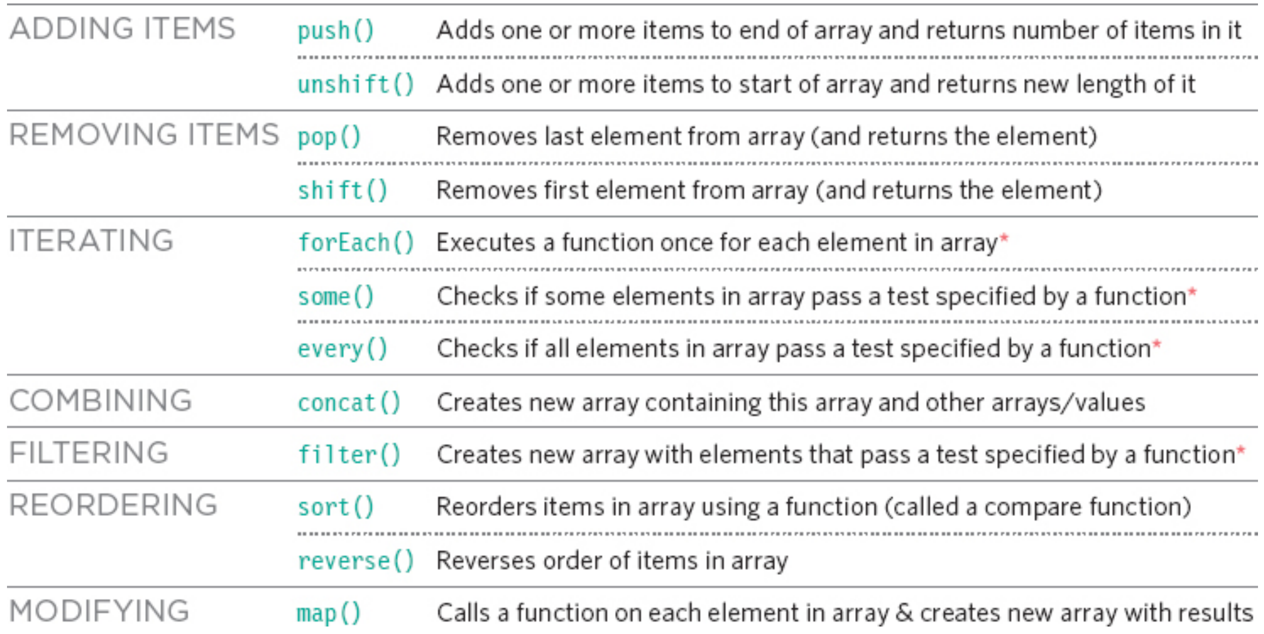
The other benefit of using arrow functions with promises/callbacks is that it reduces the confusion surrounding the this keyword. In code with multiple nested functions, it can be difficult to keep track of and remember to bind the correct this context. In ES5, you can use workarounds like the .bind method (which is slow) or creating a closure using var self = this;

```
// ES5
API.prototype.get = function(resource) {
var self = this;
return new Promise(function(resolve, reject)
{ http.get(self.uri + resource, function(data)
{resolve(data); });
        });
};
```

```
// ES6
API.prototype.get = function(resource) {
return new Promise((resolve, reject) => {
http.get(this.uri + resource,
function(data) {
                              resolve(data);
                        });
              });
};
```

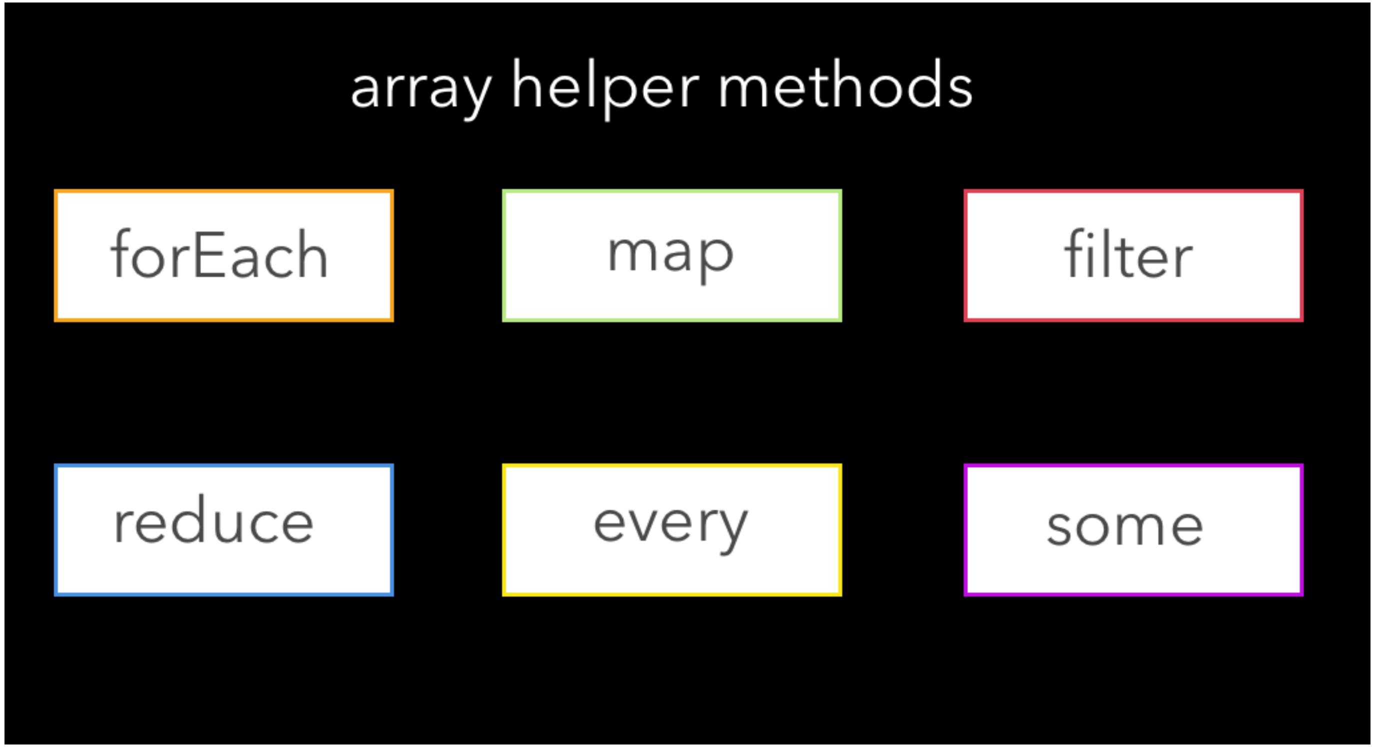
https://www.sitepoint.com/es6-arrow-functions-new-fat-concise-syntax-javascript/ Useful link

Array methods

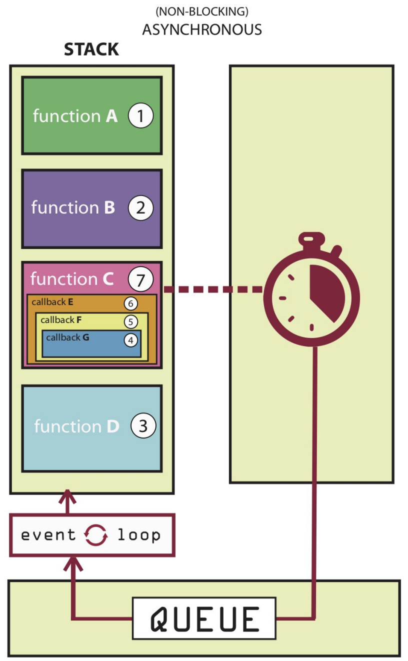


ADDING ITEMS	`push()`	Adds one or more items to end of array and returns number of items in it
	`unshift()`	Adds one or more items to start of array and returns new length of it
REMOVING ITEMS	`pop()`	Removes last element from array (and returns the element)
	`shift()`	Removes first element from array (and returns the element)
ITERATING	`forEach()`	Executes a function once for each element in array*
	`some()`	Checks if some elements in array pass a test specified by a function*
	`every()`	Checks if all elements in array pass a test specified by a function*
COMBINING	`concat()`	Creates new array containing this array and other arrays/values
FILTERING	`filter()`	Creates new array with elements that pass a test specified by a function*
REORDERING	`sort()`	Reorders items in array using a function (called a compare function)
	`reverse()`	Reverses order of items in array
MODIFYING	`map()`	Calls a function on each element in array & creates new array with results

Array methods



JS is async?

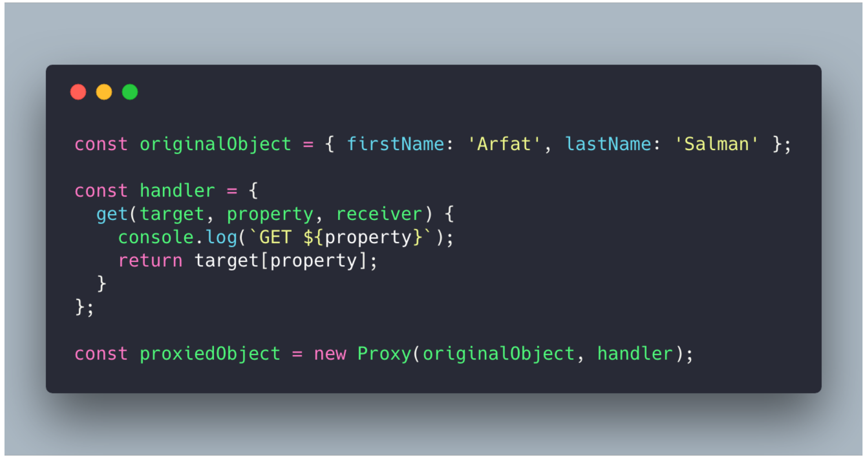


According to Wikipedia: *Asynchrony in computer programming refers to the occurrence of events independently of the main program flow and ways to deal with such events.*
In programming languages like e.g Java or C# the "main program flow" happens on the main thread or process and "the occurrence of events independently of the main program flow" is the spawning of new threads or processes that runs code in parallel to the "main program flow".
This is not the case with JavaScript. That is because a JavaScript program is single threaded and all code is executed in a sequence, not in parallel. In JavaScript this is handled by using what is called an *"asynchronous non-blocking I/O model"*.

https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff

Proxy

The **Proxy** object is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc).

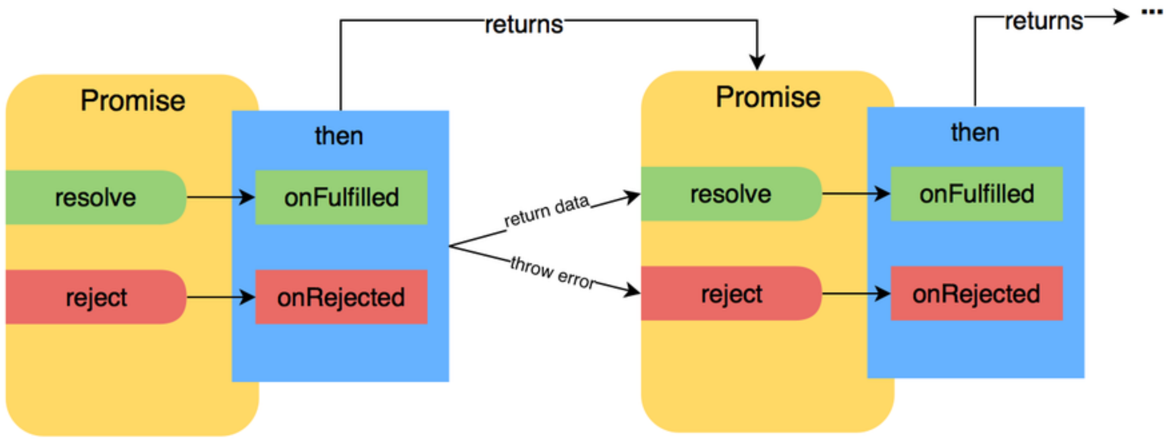


```
const originalObject = { firstName: 'Arfat', lastName: 'Salman' };

const handler = {
  get(target, property, receiver) {
    console.log(`GET ${property}`);
    return target[property];
  }
};

const proxiedObject = new Proxy(originalObject, handler);
```

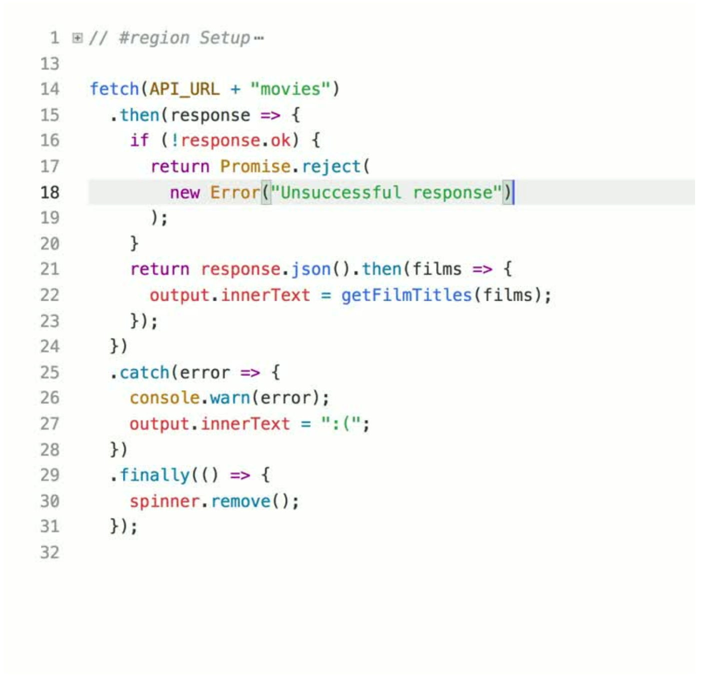
```
var handler = {
get: function(target, name) {
return name in target ?
        target[name] : 37;
        }
};
var p = new Proxy({}, handler);
p.a = 1;
p.b = undefined;
console.log(p.a, p.b); // 1,
undefined
console.log('c' in p, p.c); //
false, 37
```

Promise

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.



Promise



```
// #region Setup⋯

fetch(API_URL + "movies")
  .then(response => {
    if (!response.ok) {
      return Promise.reject(
        new Error("Unsuccessful response")
      );
    }
    return response.json().then(films => {
      output.innerText = getFilmTitles(films);
    });
  })
  .catch(error => {
    console.warn(error);
    output.innerText = ":(";
  })
  .finally(() => {
    spinner.remove();
  });
```

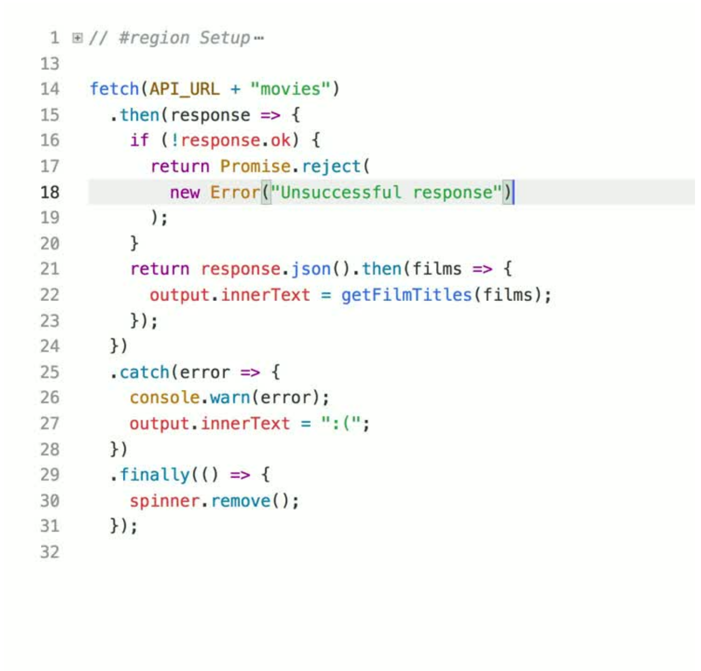
The Promise object in JavaScript offers a few useful built-in methods, with Promise.all and Promise.race being two such methods. Even though these two methods both take arrays of promises as argument, there's a big difference

between Promise.all vs Promise.race.

Both of the Promise methods receive an array of promises, however, you'll want to choose one over the other depending on what you need to accomplish.

- **Promise.all** accepts an array of promises, and will attempt to fulfill all of them. Exits early if just 1 promise gets rejected.
- **Promise.race** also accepts an array of promises, but returns the first promise that is settled. A settled promise can either be resolved or rejected.

Async \ Await



```
// #region Setup…

fetch(API_URL + "movies")
  .then(response => {
    if (!response.ok) {
      return Promise.reject(
        new Error("Unsuccessful response")
      );
    }
    return response.json().then(films => {
      output.innerText = getFilmTitles(films);
    });
  })
  .catch(error => {
    console.warn(error);
    output.innerText = ":(";
  })
  .finally(() => {
    spinner.remove();
  });
```

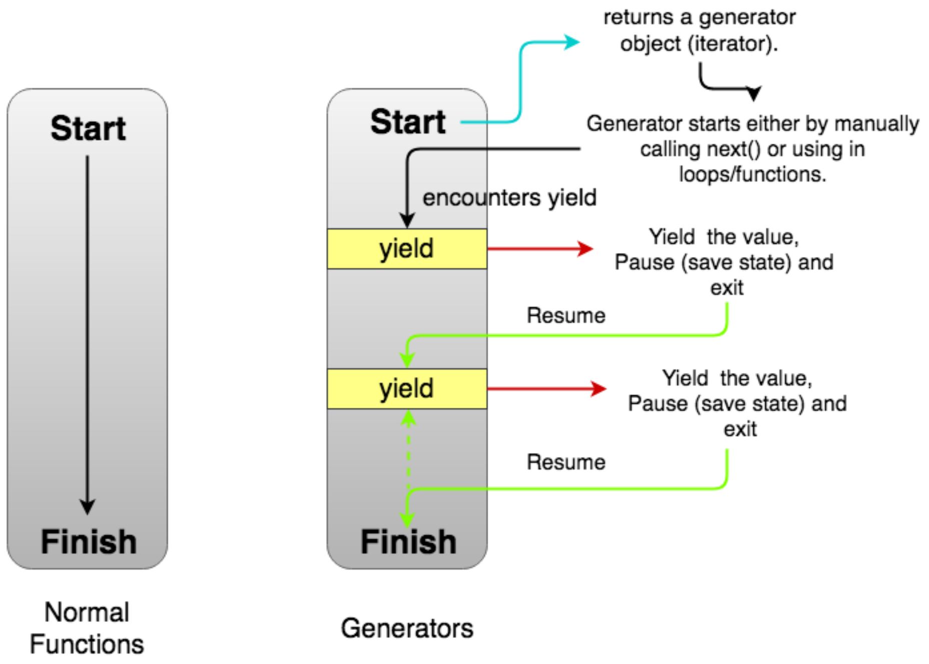
Thenable objects and high-level usage – to discuss

Async/Await is the next step in the evolution of handling asynchronous operations in JavaScript. It gives you two new keywords to use in your code: “async” and “await”. Async is for declaring that a function will handle asynchronous operations and await is used to declare that we want to “await” the result of an asynchronous operation inside a function that has the async keyword.

A basic example of using async/await looks like this:

```
async function
getSomeAsyncData(value){
        const result = await
        fetchTheData(someUrl, value);
        return result;
}
```

Generators

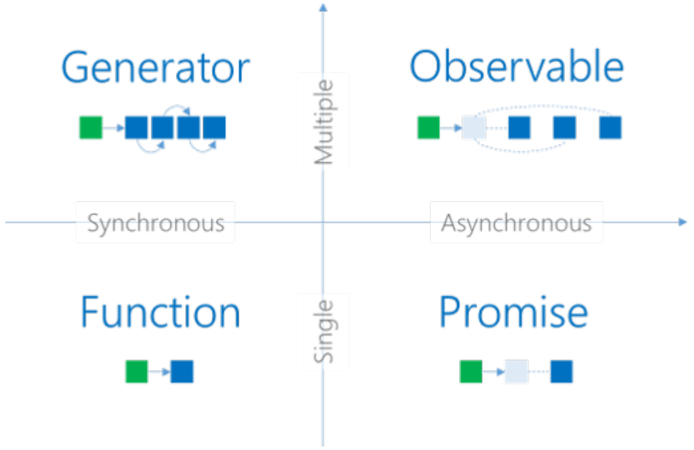

Start
Finish
Normal Functions
Start
returns a generator object (iterator).
Generator starts either by manually calling next() or using in loops/functions.
encounters yield
yield
Yield the value, Pause (save state) and exit
Resume
yield
Yield the value, Pause (save state) and exit
Resume
Finish
Generators

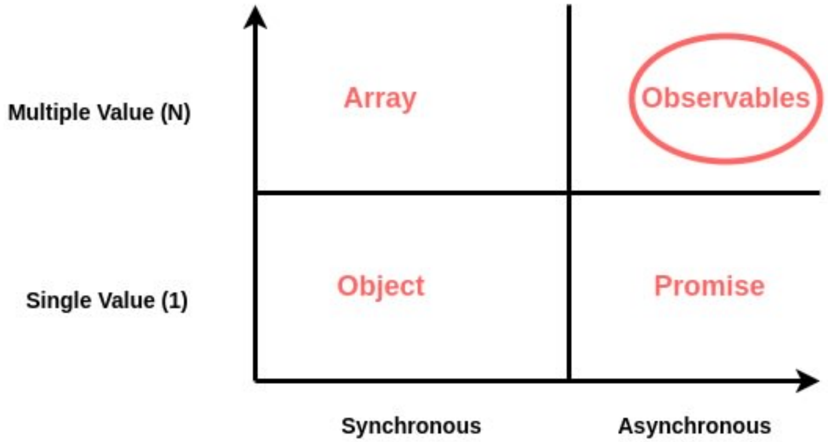
```
function* naturalNumbers() {

let num = 1;

while (true) {

yield num;

num = num + 1

}
}

const numbers = naturalNumbers();

console.log(numbers.next().value);
console.log(numbers.next().value);
```

Observables \ Rxjs

Reactive programming is a programming paradigm for writing code, mainly concerned with **asynchronous data streams.** Just a different way of building software applications which will "react" to changes that happen instead of the typical way of writing software where we explicitly write code (aka "imperative" programming) to handle those changes.

Observables are lazy Push collections of multiple values.

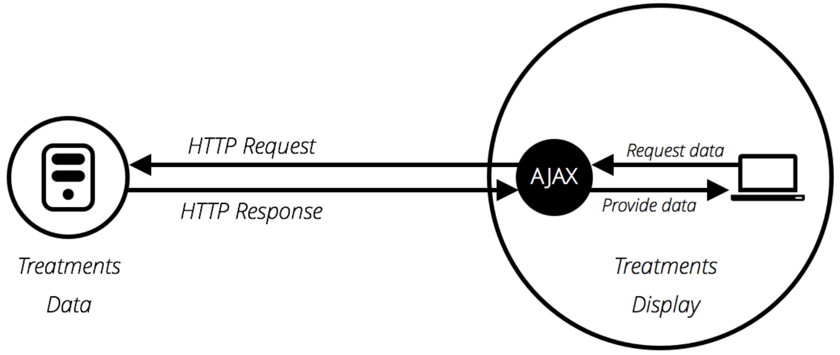




Ajax - XMLHttpRequest / Fetch

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.

This kind of functionality was previously achieved using XMLHttpRequest. Fetch provides a better alternative that can be easily used by other technologies such as Service Workers. Fetch also provides a single logical place to define other HTTP-related concepts such as CORS and extensions to HTTP.



THANKS
AND KEEP ROCKING