Modern Javascript

**Module1:**

* Describe the current state of JavaScript development.
* List the different versions of ECMAScript.
* Explain how to locate and interpret results from the ECMAScript Compatibility table.
* Utilize PlayCode to execute JavaScript code.

1 and 2



3:

http://kangax.github.io/compat-table/es6/

compatibility is in top bar

4:

<https://playcode.io/>.

**Module2:**

https://trailhead.salesforce.com/content/learn/modules/modern-javascript-development/explore-new-syntax-in-javascript-es6

* Describe the difference between function and block scoping.
* State a reason why the const keyword should be used.
* Recognize shorthand ES6+ syntax used to initialize variables.
* Identify the new destructuring syntax used to separate data.
* Identify the backtick character used to create template literals.

1 and 2:

 const no deja reasignar ni declarar variable de nombre igual porque es constante, let deja reasignar pero no puedes declarar otra variable de nombre igual, var si te deja declarar y reasignar (un block es codigo dentro de curly braces { } var no soporta block scoping por eso deja reasignar)

\*as const cant be reasigned you need to initialize it when reassigned

3 and 4:

Before:

let firstName = ‘John’, lastName = ‘Doe’;

let user = {

firstName: firstName,

lastName: lastName

{

Console.log(user);

-----now:

let firstName = ‘John’, lastName=‘Doe’;

let user = { firstName, lastName }

console.log(user);

object properties and values dont need to be repeated with ES6

DESTRUCTURING

getting values from array:

let numbers = [1,2,3,4];

let one = numbers[1], two = numbers[2];

console.log(two);

=3

with destructuring:

let [one, two, three] = numbers;

console.log(three);

=3

5:

let user = **'Me'**;

console.**log**(`The following **'user'**, ${user} has encountered an **"error"**.`);

**MODULE 3:**

https://trailhead.salesforce.com/content/learn/modules/modern-javascript-development/understand-javascript-functions

* Recognize the fat arrow syntax for functions.
* Describe the scope used with the this keyword.
* Explain why defining optional parameters in ES6+ results in cleaner code.
* Describe the different uses for the '...' operator.

1: fat arrow

defining function before:

let result = function (*i*,*j*) {

return i+j;

}

console.log(result(2,3));

defining function with arrow function

let result = (*i*,*j*) => i+j;

console.log(result(2,3));

with curly braces you need return

let result = (*i*,*j*) =>

{

return i+2;

}

console.log(result (2));

2:

this, often referred to as the “dynamic this,” which refers to the object used to invoke the function.

let message = {

hello : **'Hello'**,

names : [**'Sue'**, **'Joe'**],

showMessage: **function**() {

this.names.**forEach**(**function**(name) {

console.**log**(this.hello + **' '** + name);

});

}

}

message.**showMessage**();

in this case hello variable dont exists inside the foreach because of scope

let message = {

hello : **'Hello'**,

names : [**'Sue'**, **'Joe'**],

showMessage: **function**() {

let self = this;

this.names.**forEach**(**function**(name) {

console.**log**(self.hello + **' '** + name);

});

}

}

message.**showMessage**();

in this case hello exists because object (this) declared as self variable

\*\*

arrow functions have implicit declared object inside scope (lexical scope built in) we dont have to declare “this” again

let message = {

hello : **'Hello'**,

names : [**'Sue'**, **'Joe'**],

showMessage: **function**() {

this.names.**forEach**(name => {

console.**log**(this.hello + **' '** + name);

});

}

}

message.**showMessage**();

\*\*\*this es el objeto usado para invocar la funcion. dentro del scope de un for debe declararse denuevo sino no lo podra ver

3:

* Explain why defining optional parameters in ES6+ results in cleaner code

Rest and Spread (…) makes code cleaner as arrays can be spread into objects. but objects cant be spead into arrays

let array1 = [**'one'**, **'two'**];

let array2 = [**'three'**, **'four'**];

array1 = [...array1, ...array2];

console.**log**(...array1);

Also we can simulate named parameters by utilizing object destructuring syntax as second parameter:

function **showMessage**(who, {p1 = **"Hello"**, p2 = **"World"**} = {}) {

console.**log**(who + **' says '** + p1 + **' '** + p2);

}

**showMessage**(**"Trailhead"**); //Displays "Trailhead says Hello World"

**Module 4**

https://trailhead.salesforce.com/content/learn/modules/modern-javascript-development/work-with-classes

Work with Classes

**Learning Objectives**

After completing this unit, you'll be able to:

* Explain what's different about how you create and invoke classes in ES6+.
* List the commonly used members of a class.
* Distinguish between a base and derived class.

1:

* **Explain what's different about how you create and invoke classes in ES6+.**

before you needed to use prototypes, now class word was added although it still works with prototype inheritance instead of classes as OOP

also before:

function declarations can be hoisted. In other words, you can call a function that has yet to be declared.

Now:

Classes do not allow this sort of thing. A class can only be accessed after its definition is evaluated.

before: class looks like a function except for the prototype word that means its really a Class “Animal”

function Animal(name) {

this.name = name;

}

Animal.prototype.printName = **function**() {

console.**log**(this.name);

}

let duck = new Animal(**'duck'**);

duck.**printName**(); // Displays "duck"

After: word class was added and also constructor to appear like a class-based language but isnt because continue using prototype inheritance. the class can now be defined like this:

class Animal {

**constructor**(name) {

this.name = name;

}

**printName**() {

console.**log**(this.name);

}

}

2:Members of a class:

* List the commonly used members of a class.

|  |  |  |
| --- | --- | --- |
| Constructor | The constructor is executed automatically when creating a new instance of the class. It guarantees that an initialization function is always called. This helps maintain a valid state for the class. But you don't have to create a constructor. If one is not included, then the JavaScript engine creates an empty one for you. | constructor(name) { |
| this.name = name; |
| } |
|  |
| Static Methods | Static methods are not part of any instance of the class, meaning that you can refer to these methods without referring to an instance. The concept of static class members is not new to ES6, but the static keyword is. Prior to ES6, you had to put any methods you wanted static in the constructor. Now you can put them wherever you want in the class and just use the static keyword. | static methodName() { |
| return 'something'; |
| } |
|  |
| Prototype Methods | These methods do not include the static keyword and must be referenced with an instance. | printName() { |
| console.log(this.name); |
| } |
|  |
| Getters and Setters | These accessor functions work just like object literals and work the same as they did in ES5. Essentially you just put the get and set keywords in front of the property name. If you define a getter without a setter, then the property becomes read-only. | get area() { |
| return this.height \* this.width; |
| } |
|  |
| set area(value) { |
| this.area = value; |
| } |

3:

* Distinguish between a base and derived class.

The difference is the extends keyword. Derived classes (also known as subclasses) have them, and base classes don't. Take, for example, the following base class named Parent.

Base class without “Extends:

class Animal {

constructor(*name*) {

this.name = name;

}

getName() {

return this.name;

}

}

derived class in “Extends”

class Puppy extends Animal {

constructor(*name*) {

*super* (name);

}

getMessage() {

return 'Hello ' + *super*.getName();

}

}

let someone = new Puppy('kitten');

console.log(someone.getMessage());

Class can also be defined using expressions:

const myAnimal = class Animal {

constructor(*name*){

this.name = name;

}

printName(){

console.log(this.name);

}

}

let Dog = new myAnimal('Dog');

Dog.printName();

**Module 5**

https://trailhead.salesforce.com/content/learn/modules/modern-javascript-development/organize-code-with-modules

Organize Code with Modules

**Learning Objectives**

After completing this unit, you'll be able to:

* Describe how support for modules has evolved over the years.
* Recognize the basic syntax used to define modules.
* Distinguish between different importing styles.
* Demonstrate how named exports results in read-only properties.

1: now you use export and import to include things separated in modules

2y3: if you want to change a variable name you use “as”:

import { msg2, msg1 as msg3 } from './module2.js';

**printMsg**(msg3 + msg2);

4:

\*\*you just import the name, you cant change the variable value

The thing to remember here is that you are not allowed to reassign the exported value. It can only be changed from inside the module it was exported from

——————

**Module 6**

https://trailhead.salesforce.com/content/learn/modules/modern-javascript-development/write-asynchronous-javascript

Write Asynchronous JavaScript

**Learning Objectives**

After completing this unit, you'll be able to:

* Describe a common JavaScript pitfall known as the Pyramid of Doom.
* Recognize the commonly used structure of a promise.
* Demonstrate how asynchronous calls can be chained together using promises.
* Demonstrate how an async function can be used to call a promise.

1:

* Describe a common JavaScript pitfall known as the Pyramid of Doom.

JavaScript is single threaded, which means that only one function can run at a time.

Therefore, coding in JavaScript inevitably means working with asynchronous code. This is especially true when you need to do anything that involves I/O (input/output). Like getting data from a database, making a call to an API, or even just waiting for input from a user. Because any functions that do these types of things will undoubtedly block the browser.

A callback is just a function that executes after another function has finished executing.

function using callback:

function **doSomething**(msg, callback){

**setTimeout**(

function () {

console.**log**(msg);

**callback**();

},

1000);

}

**doSomething**(**"1st call"**, **function**() {});

piramid of doom:



2:

* Recognize the commonly used structure of a promise.

before:

with promises, The actual doSomething function (compared to the one with callback without promises), now returns a new Promise object. And instead of calling the callback, the function now calls resolve.

function doSomething(msg){

  return new Promise(

    function (resolve, reject) {

      setTimeout(

        function () {

          console.log(msg);

          resolve();

        },

        1000);

    });

}

doSomething("1st Call")

  .then(function() {

    return doSomething("2nd Call");

  })

  .then(function() {

    return doSomething("3rd Call");

});

———

3:

* Demonstrate how asynchronous calls can be chained together using promises.

-multiple promises with try catch,

function doSomething(msg){

  return new Promise((resolve, reject) => {

      setTimeout(

        () => {

          try {

            throw new Error('bad error');

            console.log(msg);

            resolve();

          } catch(e) {

            reject(e);

          }

        },

        1000);

    })

}

doSomething("1st Call")

  .then(() => doSomething("2nd Call"))

  .then(() => doSomething("3rd Call"))

  .catch(err => console.error(err.message));

4:

* Demonstrate how an async function can be used to call a promise.

* The Promise object includes four methods that you may want to check out for more advanced promise scenarios. They include:
  + **Promise.all(iterable)**—Returns promise only after all the promises in the iterable have resolved or any are rejected.

If the returned promise resolves, it is resolved with an aggregating array of the values from the resolved promises, in the same order as defined in the iterable of multiple promises.

If it rejects, it is rejected with the reason from the first promise in the iterable that was rejected.

* + **Promise.race(iterable)**—Returns promise after the first promise in the iterable has resolved or rejected.
  + **Promise.resolve(value)**—Returns promise that is resolved with the value passed in as a parameter.
  + **Promise.reject(reason)**—Returns a promise that is rejected with the reason passed in as a parameter.

**Module 7**

https://trailhead.salesforce.com/content/learn/modules/modern-javascript-development/test-your-javascript

Test Your JavaScript

**Learning Objectives**

After completing this unit, you'll be able to:

* Explain why behavioral-driven tests are unique.
* Identify the different elements used in a Jasmine testing script.
* Create a simple Jasmine test suite and run it stand-alone.

1:

* Explain why behavioral-driven tests are unique.

Behavior-driven development (BDD) is a process that was born out of Test-driven development (TDD). BDD involves organizing tests such that their behavior is tested, rather than their implementation. When designing these types of tests, think, “How can I describe, in sentence form, what this code does and what I should expect from it.”

2:

* Identify the different elements used in a Jasmine testing script.

 Jasmine you use a describe function to create a test suite, but you're really just creating a JavaScript function. Inside of that function, use the it function to specify one or more specs (or tests), which are also just functions. Inside of the spec function, put assertions to check whether the test worked.

This is what a basic test suite looks like with both a positive and a negative test:

**describe**(**"A test suite is just a function"**, **function**() {

**it**(**"This is a positive test"**, **function**() {

**expect**(true).**toBe**(true);

});

**it**(**"This is a negative test"**, **function**() {

**expect**(false).not.**toBe**(true);

});

});

=======================

=======================

=======================

Andrei Basic Javascript:

**making a calculator:**

while (option !== (0 || 1 || 2 || 3)) {

alert(“ Choose an option for calculator  0: sum, 1: substract, 2: multiply, 3: divide ");

var option = prompt(“which option are you going to take?:")

var num = prompt("you need to put number a and number b, put your number A:");

var num2 = prompt("you need to put number a and number b, put your number B:");

if (Number(option) === 0) {

var sum = Number(num) + Number(num2);

alert("the sum of both numbers is: " + sum);

} else if (Number(option) === 1) {

var substract = Number(num) - Number(num2);

alert("the sum of both substract is: " + substract);

} else if (Number(option) === 2) {

var multiply = Number(num) \* Number(num2);

alert("the sum of both multiply is: " + multiply);

} else if (Number(option) === 3) {

var multiply = (Number(num)) / (Number(num2));

alert("the sum of both multiply is: " + multiply);

} else {

alert("no elegiste una opcion valida");

break;

}

}