Taller de Ejercicios JS

**Programación web 1**

Profe. Jossy Tello

Aplicar los fundamentos del lenguaje Javascript realizando los siguientes ejercicios.

1. ***Investigue que es ecmascript y cuales son sus actualizaciones recientes.***

***ECMAScript***

La actual versión de ECMAScript es la número 7 (aunque sea la 6ª que sale realmente)

Es el estándar que define cómo debe de ser el lenguaje JavaScript. Sin entrar en datos históricos, que puedes consultar otras fuentes como la Wikipedia sobre ECMAScript, cabe decir que es la especificación que los fabricantes deben seguir al crear intérpretes para JavaScript.

JavaScript es interpretado y procesado por multitud de plataformas, entre las que se encuentran los navegadores web, así como Node JS y otros ámbitos más específicos como el desarrollo de aplicaciones para Windows y otros sistemas operativos. Todos los responsables de cada una de esas tecnologías se encargan de interpretar el lenguaje tal como dice el estándar ECMAScript.

La primera versión del estándar Ecmascript, denominada ES1, data de 1997 y procuraba definir un lenguaje apropiado para dotar de más dinamismo a la web, por entonces muy sosa. El lenguaje ha evolucionado a lo largo de estos años dando lugar a uno de los lenguajes más utilizados en el mundo del desarrollo de software.

En 2015 se libera la sexta versión del lenguaje, denominada ES6 o Ecmascript 2015, que añadió una serie de características muy importantes que eran demandadas cada vez más por los programadores. Esas características supusieron un cambio importante en el lenguaje y son extensamente utilizadas por todos los frameworks actuales y han sido incorporadas a los entornos de javascript modernos, tanto en los navegadores como en node.js.

En 2016 el comité de la ECMA encargado de la estandarización del lenguaje ha lanzado la versión ES7/ES2016. Se trata de una actualización que añade pocos cambios, al menos en comparación con los que introdujo ES6 respecto a ES5 y obedece a una nueva política de lanzamientos de versiones anuales. La finalidad de esta nueva política es evitar grandes saltos como los que se produjeron en la versión ES6.

Aunque suena muy bien eso de los pequeños cambios incrementales anuales, eso no significa que los desarrolladores tengamos instantáneamente acceso a las nuevas características que la especificación ha lanzado. La única manera de ejecutar código JavaScript es a través de un entorno de ejecución, y hasta que estos no incorporen las nuevas características no podremos usarlas.

Actualmente, y desde el año 1997, si queremos hablar con propiedad, hemos de decir que Javascript es la implementación más conocida de una especificación de lenguaje de programación denominada Ecmascript. Dicha especificación ha sido desarrollada por la [European Computer Manufacturer Asociation](https://www.ecma-international.org/) (ECMA), una asociación cuya finalidad es la definición de estándares en el mundo de la información y la comunicación, y se corresponde con el estándar [ECMA-262](https://www.ecma-international.org/publications/standards/Ecma-262.htm) de dicha asociación.

Esto no siempre fue así. En sus inicios javascript fue un novedoso lenguaje que no seguía ningún estándar. De hecho, Ecmascript surge después de que la primera versión de javascript fuese incorporada al navegador Netscape Communicator 2.0 en 1995. Dado la importancia que tiene la portabilidad de las aplicaciones distribuidas por internet, la propia empresa Netscape envió una propuesta con la referencia del lenguaje con el fin de convertirlo en un estándar al que todos los navegadores que incluyeran un entorno de ejecución de javascript tuvieran que adherirse.

1. ***Escoja mínimo 5 novedades ecmascript y explique cada novedad en 2 ejemplos.***

# Template literals

Se trata de una expresión que permite combinar cadenas multilíneas con valores de variables, las cuales se especifican dentro de la cadena usando el carácter ` (backstick) como delimitador y metiéndolas dentro de la expresión ${}, tal y como se muestra a continuación.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

var texto: "Superman";

var template = `<div>El héroe se llama ${texto} </div>

<button>Click me!</button>

<div>Esto es una capa</div>

<input type="text" value="nombre" />`

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Facilitan muchísimo la manipulación de cadenas y resultan muy útiles cuando tenemos que rellenar un texto parametrizado por variables. Un ejemplo claro es un texto que representa un trozo de código HTML. Al ser multilínea la visibilidad mejora y facilita su interpretación.

# Ámbito de variables en el bloque: let

Al igual que el keyword var se usa para declarar variables. La diferencia es que var tiene el ámbito de la función (function scoped) mientras que let tiene como ámbito el bloque (block scoped).

El siguiente código muestra el efecto de esta diferencia claramente. El segundo bucle for, declara una variable j que solo se reconoce en su bloque. Mientras que el primero, al usar var, se reconoce a lo largo de la función en que se haya escrito el bucle.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| for (var i = 0; i < 10; i++) {  // algo  }  console.log(i) // 10!  for (let j = 0; j < 10; j++) {  // algo  }  console.log(j) // j is not defined |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
|  |

# Variables que no cambian: const

También sirven para declarar variables, como let o var, y el ámbito es el bloque, como let. La diferencia es que las variables declaradas con const no pueden ser modificadas; una vez que se asigna un valor, se quedan con él hasta el final del ámbito.

Si por ejemplo hacemos esto:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| const c = 4;  c = 5;  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

El intérprete de javascript arrojará un error cuando lo ejecuta:

|  |
| --- |
| Type Error: Assignment to constant variable. |

Si sabes que una variable no debe cambiar en todo el ciclo de vida de la aplicación, es muy conveniente que la declares como const, ya que el intérprete te avisará con este error si, por equivocación, la has modificado en alguna parte del código.

# Arrow functions

Es una de las  nuevas características más útiles de ES6 y de las más utilizadas en todos los frameworks y aplicaciones modernas. Se trata de una manera más compacta de declarar funciones anónimas. Esta es la forma tradicional, que por supuesto sigue siendo válida:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| let mayor = function(a, b){       if(a > b)  return a;  else  return b;  } |

Y esta la nueva manera con funciones arrow:

|  |
| --- |
| let mayor = (a, b) => {       if(a > b)  return a;  else  return b;  }  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Mucho más compacta. La parte que interesa es la que está a la derecha del operador =, que es la definición de la función en sí. La parte de la izquierda no es más que la variable a la que es asignada la función. Esta nueva sintaxis solo valen para definir funciones como expresión, no como declaración, por tanto siempre hemos de usarla formando parte de una sentencia; una asignación o como argumento de otra función, por ejemplo.

Vemos que en la definición de una función arrow los argumentos se colocan a la izquierda del operador => entre paréntesis y separados por comas, y el cuerpo de la función se coloca a su derecha.

Si la función fuese tan corta que solo tuviera una instrucción podemos eliminar tanto llaves que acotan el cuerpo como la palabra reservada return:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| let mayor = (a, b) => a > b ? a : b; |

Y si solo tuviera un argumento, ni siquiera se necesitan los paréntesis del argumento:

|  |
| --- |
| let increment = x => x+1; |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |

Esta nueva sintaxis para declarar funciones no es sólo un syntactic sugar del lenguaje, es decir, una forma más cómoda de hacer lo mismo que antes, ya que las funciones arrow, a diferencia de las normales, no definen el parámetro intrínsecoble this. Por ello si usas this dentro del cuerpo de una función flecha siempre se referirá al this del ámbito en que haya sido definido.

# For … of

Es otro nuevo tipo de bucle de sintaxis similar al bucle for … in, pero que funciona de manera distinta. La diferencia es que mientras que for … in recorre las propiedades o índices de iterables y objetos, el bucle for … of recorre los valores de  iterables (no de objetos).

ES6 introduce un mecanismo nuevo para recorrer datos denominado iteración. La iteración se lleva a cabo usando el bucle for .. of. Son dos los conceptos centrales relativos a este mecanismo:

* Un iterable es una estructura de datos que «desea» hacer sus elementos accesibles de forma pública. Y para ello implementa un método denominado Symbol.iterator, que especifica como se debe iterar sobre la estructura.
* Un iterador es un puntero que sirve para recorrer los elementos de la estructura de datos.

Son iterables por defecto los siguientes tipos:

* Arrays
* Strings
* Maps
* Sets
* DOM data structures

Sin embargo, los objetos planos de javascript no son iterables. Un objeto plano es uno del tipo:

|  |
| --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

{"clave1": "valor1", "clave2": "valor2", ...}

Este trozo de código muestra con claridad dicha diferencia:

|  |
| --- |
| let animales = ["gato", "perro", "loro"];  for(let animal in animales){  console.log(animal); // arroja 0, 1, 2  }  for(let animal of animales){  console.log(animal); // arroja "gato", "perro", "loro"  } |

Si aplicamos for … of en un objeto de javascript, el intérprete arroja un error:

|  |
| --- |
| let usuario = { nombre: 'Alberto', apellido: 'Martín' };  for(let atributo in usuario){  console.log(atributo); // arroja nombre, apellido  }  for(let atributo of usuario){  console.log(atributo); // arroja un error-> Uncaught TypeError: usuario is not iterable} |

# Sintaxis corta de objetos

Es una forma abreviada para definir objetos que usan como nombre de la propiedad el mismo nombre de la variable que se usa para asignar su valor.

Es muy típico entre los programadores de javascript definir objetos de esta manera:

|  |
| --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

let nombre = "Juanda"

let apellido = "Rodríguez"

let usuario = {nombre: nombre, apellido: apellido};

|  |
| --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

En este código el objeto usuario tiene dos propiedades: nombre y apellido, y se asignadas al valor de unas variables que se llaman igual que la propiedad: nombre y apellido.

En ES6 esto mismo se puede escribir más escueto de la siguiente manera:

|  |
| --- |
| let nombre = "Juanda"  let apellido = "Rodríguez"  let usuario = {nombre, apellido} |

El intérprete usará los valores de las variables nombre y apellido y, como no se especifica el nombre de la propiedad, utilizará el mismo nombre de las variables.

Incluso podemos mezclar en la definición del objeto la sintaxis corta con la habitual:

|  |
| --- |
| let nombre = "Juanda"  let apellido = "Rodríguez"  let usuario = {nombre, apellido, sexo: 'varón'} |

# Rest

Es un operador que nos permite expresar el resto de los argumentos de una función con un solo identificador. El operador rest, que consiste en 3 puntos, construirá un array cuyo nombre será el del argumento al que se le aplique y cuyos elementos son todos los argumentos que se pasen en la función a partir de la posición de dicho operador.

En resumen, es un operador que empaqueta argumentos en un array. Veámoslo con un ejemplo:

|  |
| --- |
| function prueba(a, b, ...resto){  for(let r of resto){  console.log(r);  }  }  prueba(2,5,6,3,4,6); // arroja 6 3 4 6 |

Fíjate que es parecido al parámetro impícito arguments. De hecho, si solo declaramos como parámetro de la función  la variable a la que se le aplica el operador rest (los 3 puntos seguidos), este es un array que contiene los mismos valores que el argumento implícito arguments. Sin embargo no son la misma cosa, puesto que arguments, aunque se parece a un array no es propiamente un array.

Si necesitamos tratar un número indeterminado de parámetros en alguna función, por claridad, se aconseja utilizar el operador rest el lugar de arguments.

# Spread

Este operador tiene la misma sintaxis que el anterior (3 puntos seguidos) pero funcionan de manera inversa; al aplicarlo a un array, lo expande en sus valores separados por comas. De esa manera, como muestra este código, podemos usarlo para usar directamente un array como argumento de una función que requiere varios argumentos. Un código vale más que mil palabras en este caso:

|  |
| --- |
| function prueba(a, b, c){  return a + b + c;  }  let args = [1, 2, 3];  prueba(...args); // arroja 6 |
|  |

# Valores por defecto

Esta característica es muy sencilla y útil, y es algo que muchos lenguajes de programación como Python o Java ofrecen. Se trata simplemente de poder asignar en la definición de los argumentos de una función los valores que deben tomar en el caso de que en una llamada a la misma no se especifiquen.

|  |
| --- |
| function sumaNPrimerosNumeros(n = 0) {  let r = 0;  for(let i = 0; i <= n; i++){  r += i; ç  }  return r;  }  console.log(sumaNPrimerosNumeros(2)); // arroja 3  console.log(sumaNPrimerosNumeros()); // arroja 0 |

Aunque podemos declarar por defecto cualquiera de los argumentos de una función, lo normal para evitar problemas es o declararlos todos por defecto, o colocar al final de la lista de parámetros a los que vayan a tener un valor por defecto. Eso nos permite indicar, cuando nos vengan bien los valores por defecto, únicamente los parámetros que se necesitan fijar obligatoriamente.

# Destructuring

Es una forma de asignar de una sola vez varias variables con los elementos de un array u objeto.

En el caso de los array la cosa funciona así:

|  |
| --- |
| let cosas = ["lapiz", "boli", "taza"];  [x , y, z] = cosas; |

Ahora x almacena el valor “lapiz”, y el valor “boli” y z el valor “cosas”. Es muy útil cuando queremos que una función devuelva varios valores y deseamos asignar cada valor a una variable distinta de un tirón.

También podemos utilizar destructuring con objetos y asignar varias variables de una vez con los contenidos de objetos.

|  |
| --- |
| let library = {suma: (x,y) => x + y, resta: (x, y) => x -y }  {suma, resta} = library |

Ahora suma es una variable que almacenan una función para sumar y resta una variable que almacena una función para restar.

También podemos asignar solo una parte:

|  |
| --- |
| let library = {suma: (x,y) => x + y, resta: (x, y) => x -y }  {suma} = library |

Ahora suma es una variable que almacenan una función para sumar. La función para restar, sencillamente no se ha asignado.

Es importante para que esto funcione que el nombre de las variables a desempaquetar coincida con el nombre de la clave del objeto que queremos asignar.

Esto es muy útil para construir librerías que devuelven objetos y funciones. De hecho, como veremos en un momento, se utiliza para importar  objetos desde módulos externos, otra de las características de ES6.

# Clases

Javascript es un lenguaje de programación orientado a objetos. Sin embargo el estilo utilizado no es el más conocido; el cual se basa en clases. Javascript utiliza un estilo basado en prototipos, también conocido como classless (sin clases).

Según algunos es más sencillo de comprender la programación a objetos con prototipos que con clases. Sin embargo es indudable que esta última es más conocida por ser utilizada en los lenguajes de programación más conocidos como C++, Java, Python o PHP.

El soporte de clases en javascript disminuiría la curva de aprendizaje de este lenguaje a muchos programadores acostumbrados al uso de las mismas. Pues bien, a partir de ES6 queda cubierta esa demanda y podemos utilizar las clases tal y como estamos acostumbrados. Hay que decir que, en realidad se trata de un syntactic sugar. Es decir, el intérprete de javascript transforma el código que usa clases en código que usa prototipos, de manera que internamente sigue trabajando con los prototipos. Pero eso a nosotros no nos importa. Si nos sentimos confortables con las clases porque hemos aprendido a trabajar así, pues podemos usarlas sin problemas en ES6.

Estas son las características que ES6 ha incorporado al estilo de programación orientada a objetos basada en clases:

* Herencia ( extends )
* Constructores ( constructor() {} )
* Métodos de instancia ( metodo() {} )
* Métodos estáticos ( static metodo() {} )
* Overrides de métodos a hijos ( super.metodo() )
* Llamadas a constructores padre ( super() )

La declaración y estructura básica de una clase en ES6 es así:

|  |
| --- |
| class MiClase {  atributo;    metodo() {      // código  }  constructor() {  // código  }} |

Y una clase que hereda de la anterior:

|  |
| --- |
| class MiClaseHija extends MiClase {       otroAtributoNuevo;  otroMetodoNuevo(){  // código  }  constructor(){  super();  // código  }  } |

Obviamente los métodos y el constructor pueden tener argumentos, en cuyo caso hay que pasarlos a super() en el constructor de las clases derivadas.

# Módulos

Este es una de las características estrellas de ES6. En las versiones anteriores había que utilizar librerías como [requirejs](https://requirejs.org/) si queríamos organizar el código en módulos que son cargados por los scripts según se necesiten unos componentes u otros, de la misma manera que Python o Java usan import o C++ usa include.

En ES6, si queremos que un objeto, clase, función, variable o constante de un script pueda ser utilizado por otro script, le colocamos el keyword export de la siguiente manera:

|  |
| --- |
| // Archivo `miModulo.js`  export function miFuncion(){ return "hola" }  export class MiClase{ }  export let pi = 3.14 |

Cuando un archivo exporta algo o todo lo que en él se define, se considera un módulo.

Después podemos importar dichos módulos desde otros archivos para usar sus elementos exportados:

|  |
| --- |
| // Archivo otroArchivo.js  import { pi, miFuncion } from './miModulo.js' |

Observa cómo se utiliza el destructuring en la importación de objetos.

También podemos realizar la exportación en una sola instrucción:

|  |
| --- |
| // Archivo `miModulo.js`  function miFuncion(){ return "hola" }  class MiClase{ }  let pi = 3.14;  export {miFuncion, MiClase, pi} |

De esta manera, cuando el archivo es muy extenso, echando un vistazo a la línea donde ser realizar la exportación, sabemos qué cosas hemos exportado de ese módulo.

# Maps

Los mapas (Maps) y conjuntos (Sets) son objetos que sirven para tratar con colecciones de objetos.

En el caso de los mapas los integrantes de la colección están constituidos por pares de objectos en los que el primer elemento se denomina clave y el segundo valor, de manera que las claves son únicas en toda la colección. Esto significa que el acceso a los distintos valores de la colección se hace rápidamente a través de la clave.

En realidad un mapa es estructuralmente algo muy parecido a un objeto de javascript, ya que estos también tratan con conjuntos de elementos que son pares clave => valor. Sin embargo conceptualmente son diferentes: mientras que los objetos de javascript están pensados para almacenar cualquier tipo de estructura con elementos heterogéneos que conforman sus atributos y métodos y ofrecen un prototipo de funciones, los mapas están pensados simplemente para almacenar colecciones de elementos generalmente homogeneos, es decir, del mismo tipo.

Esto no significa que no podamos usar mapas para almacenar datos de distintos tipos ni que no podamos usar objetos para almacenar datos homogeneos. Pero si la intención es gestionar una colección de datos similares referenciados por una clave, es preferible usar un mapa. Si la intención es crear un objeto con sus atributos (que serán seguramente de distintos tipos), métodos y prototipos, entonces la única vía es usar un objeto.

Observa que los propios mapas son objetos de javascript, ya que todos los tipos que no son básicos (números, cadenas y booleanos) son objetos de javascript, cada cual con su prototipo. Los mapas también lo son, lo que ocurre es que tienen un prototipo con funciones diseñadas para la manipulación de colecciones de pares.

Un objeto de tipo mapa se crea mediante la función constructora Map():

let mapa = new Map();

Una vez creado lo podemos rellenar de pares con la función set():

mapa.set("pi", 3.14159265359); mapa.set("e", 2.71828182846);

Podemos comprobar si existe un elemento con una clave determinada con has():

let tiene\_pi = mapa.has("pi"); // true

Podemos acceder a cualquier elemento a través de su clave con get();

let pi = mapa.get("pi");

Obtener todas las claves con keys();

let claves = mapa.keys();

Obtener todos los valores con values();

let valores = mapa.values();

Y todos los elementos con entries();

let elementos = map.entries();

Borrar el  elemento con una clave determinada con delete();

map.delete("pi");

O borrar todos los elementos con clear();

map.clear();

[***https://www.returngis.net/tag/jshint/***](https://www.returngis.net/tag/jshint/)

[***https://programamos.es/unidad-5-caracteristicas-de-ecmascript-es6/***](https://programamos.es/unidad-5-caracteristicas-de-ecmascript-es6/)

[***https://www.returngis.net/tag/jshint/***](https://www.returngis.net/tag/jshint/)