TEMA

Tema 3. Estructuras definidas por el programador

Desarrollo de aplicaciones web

Desarrollo web en entorno cliente

Autor: Cristian Catalán

Linkia FP

Tema 3: Estructuras definidas por el programador

¿Qué aprenderás?

- Crear Arrays en JavaScript y utilizar sus métodos y propiedades.
- Crear objetos JSON y utilizarlos para estructurar la información.
- Recorrer Arrays y objetos JSON.
- Crear y utilizar objetos personalizados asignándoles propiedades y métodos.
- Depurar el código trabajando con Arrays y objetos.

¿Sabías que…?

- JavaScript es un lenguaje orientado a objetos basado en prototipos.
- En JavaScript las funciones son objetos.
- El bucle “For each” itera sobre todas las propiedades de un objeto.
- Unity puede programarse con una versión modificada de JavaScript.

3.1. Funciones

Las funciones nos permiten encapsular un fragmento de código que realiza una cierta tarea para poderla llamar fácilmente siempre que nos sea necesario. Técnicamente podemos hablar de funciones cuando no formen parte de un objeto y en el caso que formen parte de un objeto se llaman métodos. Pero a la práctica en JavaScript ésta distinción no tiene mucho sentido, ya que en JavaScript todas las funciones forman parte de algún objeto. Incluso aquellas que definamos directamente en nuestro código JavaScript serán métodos que añadiremos al objeto *window* del documento en el que se ejecute el código. Por eso es posible que nos refiramos a funciones y métodos indistintamente, aunque se suele utilizar más el término función para aquellas definidas directamente en el código y método cuando se añaden a una clase que hayamos creado.

Para crear una función en JavaScript solamente necesitamos escribir la sintaxis:

function nombreFuncion(){ }

Entre los paréntesis *"( )"* añadiremos separado por comas los parámetros que pueda recibir la función y ene las llaves *"{ } "* añadiremos el bloque de código que forme la función. En el siguiente ejemplo estamos creando una función que retorna el resultado de sumar los dos primeros parámetros y muestra el tercer parámetro por consola:

```
function ejemploFuncion(num1, num2, texto){
    console.log(texto);
    let resul = num1+num2;
    return resul;
}
```

A diferencia de otros lenguajes no es necesario indicar el tipo de retorno de la función, ya que en JavaScript cada vez que se ejecuta una función puede retornar un tipo de variable distinta (o no retornar nada). Como vemos en el ejemplo anterior tampoco hemos de indicar el tipo de dato de cada parámetro.

Para ejecutar una función es suficiente con escribir el nombre de la función y los parámetros que le queremos pasar. A diferencia de Java lo que identifica una función es únicamente su nombre, por ello no pueden existir dos funciones con igual nombre aunque tengan distintos parámetros.

Si llamamos a una función sin escribir todos sus parámetros la función los recibirá con un valor "*undefined*".

En el siguiente ejemplo llamamos tres veces a la función que hemos creado anteriormente pasando distintos tipos de parámetros. La primera vez que se ejecuta imprime por consola el texto “hola” y retorna el valor “15”. La segunda vez que se ejecuta imprime por consola el valor “undefined” ya que no se ha pasado un 3r parámetro y retorna el valor “10_texto” debido a que como el 2º parámetro es una string el operador “+” concadena los dos primeros parámetros. En el tercer caso vuelve a imprimir por consola el valor “undefined” y como el 2º parámetro también es *undefined* y no puede realizar ninguna operación con un valor *undefined* retorna un valor NaN (not a number).

```
function ejemploFuncion(num1, num2, texto){
    console.log(texto);  //hola , undefined, undefined

    let resul = num1+num2;
    return resul;
}

let resp =ejemploFuncion(10,5,"hola");
console.log(resp); //15

resp = ejemploFuncion(10,"_texto");
console.log(resp); //10_texto

resp = ejemploFuncion(10);
console.log(resp); //NaN
```

Ejemplo de paso de parámetros en funciones

3.1.1. Parámetros por defecto

Como en JavaScript no estamos obligados a llamar a una función indicándole todos los parámetros, es muy común que sea llamada con parámetros de menos. Si no controlamos esto parámetros por defecto sus valores serán “*undefined*” , lo que puede provocar un mal funcionamiento de la función. A continuación vamos a ver cómo podemos asignar valores por defecto a los distintos parámetros de una función.

Actualmente la forma más simple de asignar un valor por defecto a un parámetro es añadiendo en la definición de la función y después de cada parámetro el operador *“= “* seguido del valor que por defecto queramos asignarle.

En el siguiente ejemplo hems definido como valor por defecto del 2º parámetro el número 0 y como valor por defecto del 3r parámetro el texto “sin texto”. Para ver como funciona hemos llamado a la función dos veces. En la primera le hemos pasado los 3 parámetros y al ejecutarse muestra el texto pasado por parámetro “hola” y retornando *15*. En la segunda solo le hemos pasado el 1r parámetro, así que cogerá los valores por defecto mostrando por consola el valor “sin texto” y retornando 10.

```
function ejemploFuncion(num1, num2=0, texto="sin texto"){
    console.log(texto); //hola , sin texto
    let resul = num1+num2;
    return resul;
}

let resp =ejemploFuncion(10,5, "hola");
console.log(resp); //15

resp = ejemploFuncion(10);
console.log(resp); //NaN
```

Ejemplo de valores por defecto

En versiones anteriores de JavaScript si queríamos asignar valores por defecto debíamos comprobar manualmente si un parámetro era undefined y en tal caso darle el valor por defecto. Esta operación la podemos hacer manualmente utilizando el operador ||. Ya que como vimos si el operador OR no recibe como primer operador un valor convertible a *true* se devuelve directamente el segundo operando. Y como te puedes suponer, *undefined* no puede ser convertido a true. Así que el código anterior utilizando el operador || quedaría de la siguiente manera:

```
function ejemploFuncion(num1, num2=0, texto="sin texto"){
    num2=num2||0;
    texto = texto||"sin texto";
    console.log(texto); //hola , sin texto
    let resul = num1+num2;
    return resul;
}

let resp =ejemploFuncion(10,5, "hola");
console.log(resp); //15

resp = ejemploFuncion(10);
console.log(resp); //NaN
```

Ejemplo de valores por defecto con OR

Podemos definir una función en cualquier punto de nuestro código JavaScript, es decir, no hace falta que la definamos antes de usarla.

3.1.2. Funciones anónimas

En JavaScript también podemos definir funciones sin especificar su nombre. Es lo que llamaremos **funciones anónimas**. Como las funciones anónimas no tienen nombre que las definan no pueden ser ejecutadas a partir de su nombre. ¿Entonces para qué sirven? El truco consiste en pensar que JavaScript trata a las funciones como objetos. Y si una función es un objeto entonces puede ser almacenada en una variable para posteriormente ser ejecutada. En el siguiente ejemplo definimos la función de forma anónima y la guardamos en una variable de nombre "ejemploFuncion". Fíjate que el funcionamiento es el mismo que si le hubiéramos dado un nombre.

```
let ejemploFuncion = function (num1, num2=0, texto="sin texto"){
    console.log(texto); //hola , sin texto
    let resul = num1+num2;
    return resul;
}

let resp =ejemploFuncion(10,5, "hola");
console.log(resp); //15

resp = ejemploFuncion(10);
console.log(resp); //NaN
```

Ejemplo de valores por defecto

Incluso podemos pasar funciones por parámetro. En el siguiente ejemplo hemos definido la funcion "ejecutaParametro ()" que ejecutará el parámetro que reciba. Luego hemos guardado en una variable "ejemploFuncion" una función con valores por defecto. Finalmente al llamar a "ejecutaParametro ()" pasando la función creada se ejecutará la función anónima con los parámetros por defecto (ya que no le especificamos nuevos parámetros).

```
function ejecutaParametro(unaFuncion){
    return unaFuncion(); //5
}
let ejemploFuncion = function (num1=5, num2=0, texto="sin texto"){
    console.log(texto); //sin texto
    let resul = num1+num2;
    return resul;
}

let resp = ejecutaParametro(ejemploFuncion);
console.log(resp); //5
```

Ejemplo de función pasada como parámetro

También podemos hacer que una función retorne una función. En el siguiente ejemplo *generaFuncion()* se encarga de generar una función y retornarla. Posteriormente podemos ejecutar el valor retornado como una función.

```
function generaFuncion() {
    let fun = function (num1 = 5, num2 = 0, texto = "sin texto") {
        console.log(texto); //sin texto
        return num1 + num2;
    }
    return fun;
}
let funcionGenerada = generaFuncion()
console.log(funcionGenerada()); //5
```

Ejemplo de función que retorna otra función

Las funciones anónimas tienen la particularidad de que su definición ha de aparecer antes de ser usada en el código JavaScript.

3.1.3. Funciones flecha

Las funciones flecha ofrecen una sintaxis aún más reducida que las funciones anónimas.

Además ofrecen la particularidad de no tener un this propio y por lo tanto no "deberíamos" usarlas como métodos. Pero el hecho de no tener un this propio provoca que podamos acceder directamente a las variables definidas dentro del objeto en el que se ha creado la función. Aunque esto es algo que veremos con los objetos.

Podemos definir una función flecha con la sintaxis:

(parametro1, parametro2) => { / código de la función*/ };*

Como puedes observar las funciones flecha siempre son anónimas.

En el siguiente ejemplo vamos a definir una función anónima y una función flecha.

```
var sumaNumeros = function (x, y) {
    return x + y;
}
console.log(sumaNumeros(3, 4)); // 7

const restaNumeros = (x, y) => { return x - y; }
console.log(restaNumeros(3, 4)); // -1
```

Ejemplo de una función anónima y una función flecha.

3.2. Arrays

Las variables del tipo Array son una estructura de datos que permiten almacenar distintos valores de forma ordenada en una misma variable.

En JavaScript una Array puede almacenar variables de distintos tipos.

3.2.1. Declarar un array

Podemos declarar un array con el constructor "Array" o como si fuera un literal. Las tres sintaxis principales para declarar un array son las siguientes:

1. let array1 = new Array (longitud);
2. let array2 = new Array ("primer valor", "Segundo Valor", …);
3. let array3 = ["primer valor", "segundo Valor",….]

En las dos primeras formas utilizamos el constructor "Array", en éste caso si pasamos un único parámetro numérico nos generará un array de una longitud igual al número indicado. De ésta forma los valores que almacenará en cada posición del array tendrán un valor *undefined* por defecto.

Declarando un array con el constructor "Array" y pasándole más de un valor estaremos definiendo los valores iniciales que contendrán el array y su longitud, que será igual al número de elementos del array. Hay que recordar que en JavaScript un array puede almacenar valores de distinto tipo.

La tercera sintaxis es la correspondiente a declarar un array como un literal. Recordemos que un literal es una variable que ha sido creada sin un constructor. Ésta forma es la más rápida y más utilizada normalmente. Funciona igual que la segunda, es decir indicando los valores iniciales que queremos que contenga el array. Si inicialmente queremos que esté vacío solo tenemos que escribir las llaves sin nada dentro "[ ]";

A continuación podemos ver un ejemplo de las tres formas de declarar un array:

```
//array vacía de longitud 3 declarada con el constructor
let array1 = new Array(3);

//array con 2 elementos y longitud 2 declarada con el constructor
let array1 = new Array("primer valor",88);

//array con 2 elementos y longitud 2 declarada como literal
let  array3 =["primer valor",88];
```

tres formas de declarar un array en JavaScript

3.2.2. Contenido de un array

Una vez ya hayamos declarado el array podemos añadirle contenido. No podemos añadir contenido a un array sin haberlo declarado antes.

Podemos añadir nuevos elementos a un array indicando la posición en la que queremos añadir el nuevo elemento y asignándole el nuevo valor a guardar con la sintaxis:

miArray[posición]= nuevoValor;

Si en la posición indicada ya existía un valor, éste será sobrescrito con el nuevo valor. Sin la posición no existía se creará la nueva posición con el nuevo valor y modificándose automáticamente la longitud del array .

Podemos obtener a la longitud de un array a través de su propiedad .length con la sintaxis:

miArray.length;

Podemos modificar la longitud de un array a través de su propiedad .length con la sintaxis:

miArray.length=nuevaLongitud;

Modificar la longitud manualmente, provocara que eliminemos elementos si indicamos una longitud menor o se añadirán nuevos elementos con valor *undefined* si indicamos una longitud mayor.

En el siguiente ejemplo hemos creamos un array inicialmente de 2 elementos. Después le modificamos el valor del primer elemento y le añadimos un nuevo array en la posición 4 provocando que aumente su longitud a 5 y que se creen tres posiciones nuevas, una con el nuevo valor y las otras dos con el valor *undefined*. Finalmente le modificamos la longitud a 3 haciendo que se eliminen todos aquellos valores situados en posiciones más altas.

```
let  miArray =["primer valor",88];
console.log(miArray[0]+","+ miArray[1]); //primer valor,88
console.log("longitud:"+miArray.length); //longitud:2

//modificamos la posición 0
miArray[0] = 33;
//añadimos un nuevo valor en la posicion 4
miArray[4] = "último valor";

console.log(miArray[0]+","+ miArray[1]+","+miArray[2]+","+
miArray[3]+","+ miArray[4]);
//33,88,undefined,undefined,último valor
console.log("longitud:"+miArray.length); //longitud:5

miArray.length=3;
console.log(miArray[0]+","+ miArray[1]+","+miArray[2]+","+
miArray[3]+","+ miArray[4]);
//33,88,undefined,undefined,undefined
console.log("longitud:"+miArray.length); //longitud:3
```

Ejemplo de creación de un array, añadir nuevos elementos y reducir su longitud.

3.2.3. Recorrer un array

En JavaScript siempre encontraremos múltiples caminos para realizar las acciones que queramos. Vamos a ver a continuación dos formas de recorrer un array. La primera es la forma más clásica, utilizando un bucle *for*. Recordando que dentro de un array un valor se almacena en una posición numérica que empieza en 0 y termina con un valor numérico igual a la longitud del array menos 1, cuando queramos acceder a todos los elementos de un array podemos recorrer todos los números del 0 hasta la longitud-1.

En el siguiente ejemplo se utiliza un bucle para obtener y mostrar por consola los valores:

0:primer valor

1: 88

2: undefined

3: undefined

4: undefined

5: último valor

```
let  miArray =["primer valor",88];
miArray[5]="último valor";

for(let k=0;k < miArray.length; k++){
    console.log(k +":"+miArray[k]);
} //33,88,undefined,undefined, undefined ,último valor
```

Ejemplo de creación de un array, añadir nuevos elementos y reducir su longitud.

Como observamos en el ejemplo al recorrer un array con un *for* accedemos también a todas aquellas posiciones que contienen un valor *undefined*. Si queremos acceder solo a aquellas posiciones con valores podemos utilizar el método *forEach* definido en el mismo objeto array. Este método nos va a permitir definir una función anónima que se ejecutará por cada uno de los elementos del array. Ésta función cada vez que se ejecuta recibe como parámetro el elemento al que accede y la posición dentro del array.

En el siguiente ejemplo vamos a utilizar esta función para recorrer todo el array mostrando los valores:

0: primer valor

1: 88

5: último valor

```
let  miArray =["primer valor",88];
miArray[5]="último valor";

miArray.forEach(function (elemento, indice, array) {
    console.log(indice+":"+elemento);
});
//primer valor,88 ,último valor
```

Ejemplo de creación de un array, añadir nuevos elementos y reducir su longitud.

3.2.4. Métodos y propiedades del objeto Array

Cualquier variable del tipo Array hereda todos las propiedades y métodos que mostraremos a continuación.

La propiedad principal del objeto Array es length.

Propiedad	Descripción
`let miArray = ["Ei,3]; miArray.length;`	.length retorna el número de elementos almacenados en posiciones numéricas en el array. En éste caso mostraría el valor 2.

Los métodos principales nos permiten trabajar con el array añadiendo y eliminando elementos del array. Vamos a ver el listado de métodos teniendo presente que hemos declarado un array de la siguiente manera:

let miArray = ["pera",33,"azul"];

Método	Descripción
miArray.isArray()	Retorna true si la variable es un array y false en caso contrario. En este caso retorna *true.*
miArray.pop()	Quita el último elemento del array y lo retorna. En este caso el array quedaría con una longitud de 2, retornaría el valor “azul” y quedaría con los elementos: *["pera",33]*
miArray.push("verde")	Añade un elemento al final del array y retorna la nueva longitud. En este caso el array quedaría con una longitud de 5 y los elementos: *["pera",33,"azul","verde"]*
miArray.shift()	Elimina el primer elemento de un array y lo retorna. En este caso el array quedaría con una longitud de 2, retornaría el valor “pera” y quedaría con los elementos: *[33,"azul"]*

miArray.unshift("verde")	Añade un elemento al inicio del array y retorna la nueva longitud. En este caso el array quedaría con una longitud de 5 y los elementos: *["verde","pera",33,"azul"]*
miArray.splice(2, 1, "kiwi","mango")	*Splice* es un método muy versátil que permite situarse en una posición del array y a partir de allí eliminar y añadir elementos. Retorna los elementos que elimina (si los hay) El primer parámetro indica en qué posición del array vamos a situarnos para hacer las modificaciones. El segundo parámetro indica cuantos elementos vamos a eliminar. A continuación se pueden indicar tantos parámetros como se quieran y se añadirán como valores en el array. En el ejemplo nos situamos en la posicion 2, borramos 1 elemento y a continuación añadimos los valores “kiwi” y “mango”. Finalmente el array queda con los valores: *["pera", 33, "kiwi", "mango"]*
miArray.slice(0,2);	Retorna una copia de una parte del array empezando por la posición indicada por el primer parámetro hasta la posición indicada por el segundo parámetro (no incluido). Si no se indica el segundo parámetro se extraerá hasta el final del array. Si el primer parámetro es negativo, la posición inicial se contará desde el último elemento (si se indica un -1 se extraerá solo el último elemento). Si el segundo parámetro es negativo, la posición final se contará desde el último elemento (si se indica un -1 se extraerá hasta el penúltimo elemento). En éste caso retornaría un nuevo array con los valores: *["pera", 33]*

miArray.reverse()	Invierte el orden de los valores que forman el array. El array quedaría con los valores: *["azul",33,"pera"];*
miArray.join("-+")	El método *join* retorna una string como resultado de convertir a string todos los valores que forman el array y concadenarlos con el valor pasado por parámetro. En éste caso retornaría la string: *"pera-+33-+azul"*
miArray.concat([55,"limon"],["66"])	Retorna un nuevo array como resultado de concadenar el array con todos los demás arrays pasados como parámetro. En éste caso retornaría un nuevo array con los valores: *["pera", 33, "azul", 55, "limon", 66]*
let miArray2 = ["a",9,"a",9,"a",9]; miArray2.indexOf("a", 0), //0 miArray2.indexOf("a", 1), //2	Retorna la posición en la que se encuentra un valor empezando recorriendo el array hacia delante desde una posición inicial. El primer parámetro indica el valor que se quiere buscar y el segundo la posición a partir de la que se quiere buscar. En el primer ejemplo se empieza a buscar el valor "a" desde la posición 0 y se encuentra en la misma posición 0. En el segundo ejemplo se empieza a buscar el siguiente valor "a" a partir de la posición 1 y se encuentra en la posición 2.

let miArray2 = ["a",9,"a",9,"a",9]; miArray2.lastIndexOf("a"), //4 miArray2.lastIndexOf("a", 3), //2	Retorna la última posición en la que se encuentra un valor recorriendo el array hacia atrás desde una posición inicial. El primer parámetro indica el valor que se quiere buscar y el segundo la posición a partir de la que se quiere buscar hacia atrás. En el primer ejemplo se empieza a buscar el valor "a" desde la última posición y se encuentra en la posición 4. En el segundo ejemplo se empieza a buscar hacia atrás el siguiente valor "a" a partir de la posición 3 y se encuentra en la posición 2.

3.3. Arrays multidimensionales

En JavaScript los arrays multidimensionales son array en los que alguno de sus elementos es otro array. Esto provoca que a diferencia de Java los arrays multidimensionales no sean "cuadriculados" y por lo tanto que recorrer todos los elementos de un array multidimensional pueda llegar a ser muy complejo (pues no todos los elementos de la primera dimensión de un array multidimensional han de ser arrays).

Para acceder a un valor en un array multidimensional deberemos de indicar la posición del valor en la primera y la segunda dimensión siguiendo la sintaxis:

nombreArray[posicionPrimeraDimension] [posicionSegundaDimension]

Podemos crear un array multidimensional en JavaScript mediante dos posibles sintaxis:

1. Definiendo un array normal y añadiéndole nuevos arrays.

```
var contactos1 = [];
contactos1[0] = "lista contactos"
contactos1[1] = ["Toni", 33];
contactos1[2] = new Array("Lu", 45);
contactos1[3] = ["Gin", 66];
console.log(contactos1[0],"-",contactos1[1][0],"-",contactos1[3][1]);
//lista contactos - Toni - 66
```

En este caso se mostrara por consola el elemento ubicado en la posición 0, la posición 1,0 y la posición3,1 que se corresponden con los valores: *lista contactos - Toni - 66*

2. La segunda sintaxis para crear un array consiste en crear un array multidimensional en la misma definición del array indicando que los valores que va a tener el array son arrays:

```
var contactos = ["lista contactos",["Toni", 33],["Lu", 45] ,["Gin", 66]]
console.log( contactos[1].length ,"-"
,contactos[contactos.length-1][contactos[contactos.length-1].length-1]);
```

Este ejemplo genera un array con los mismos valores que el generado en el caso anterior. En este ejemplo vemos cómo podemos acceder a la propiedad *length* de un subarray. En éste caso se mostrará por consola el valor *2* y *33* correspondientes respectivamente a la longitud del primer subarray almacenado en la posición 1 y el valor almacenado en la última posición del último array almacenado en *contactos.*

En ambo casos vemos como un array multidimensional puede contener otros valores que no sean un array, en este caso el texto "lista contactos". Esto provoca que recorrer un array multidimensional del que no se conozca su estructura pueda llegar a ser muy complejo.

3.3.1. Recorrer un array multidimensional

Como siempre que queramos recorrer un array multidimensional, necesitaremos un bucle anidado. Pero para asegurarnos que no intentamos acceder a un valor como si fuera un array cuando en realidad es otra cosa, vamos a utilizar el operador *instanceof* para comprobar por cada elemento si se trata de un Array. En caso afirmativo ejecutaremos un bucle anidado que imprimirá todos sus valores. El resultado del siguiente código será:

0>lista contactos
1:0>Toni
1:1>33
2:0>Lu
2:1>45
3:0>Gin
3:1>66

```
var contactos = ["lista contactos",["Toni", 33],["Lu", 45] ,["Gin", 66]]
contactos.forEach(function (elemento, indice) {//primer bucle
      if(elemento instanceof Array){ //miramos si el valor es un array
          elemento.forEach(function(subElem,subIndex){ // bucle anidado
          console.log(indice+":"+subIndex+">"+subElem);
        }
        );
    }else{
        console.log(indice+">"+elemento); //0>lista contactos
    }
});
```

En el siguiente video podemos ver como crear y recorrer arrays y depurar errores derivados en JavaScript.

Video: Crear y recorrer arrays y depurar errores

3.4. Paradigma de la programación en entorno cliente

Aunque JavaScript es un lenguaje basado en objetos, no utiliza el concepto de programación orientado a objetos (POO) igual que Java aunque tiene muchas similitudes. Conscientes de que la mayoría de programadores trabajan con el paradigma de POO , cada vez más el estándar trabaja para aumentar su compatibilidad con la POO típica de Java.

Comparándolo con Java podríamos decir que el concepto de "Clase" en JavaScript lo llamaremos "Constructor" y su superclase global es el constructor "*Object*".

En JavaScript un objeto es una variable que contiene una colección de propiedades. Cada propiedad es una asociación entre un nombre y un valor. Ese valor puede ser cualquier valor que pudiéramos asociar a una variable. Así que los métodos de un objeto serán aquellas propiedades que tengan asociadas una función.

En JavaScript podemos crear un objeto de tres formas distintas:

- Creando un *objeto* del tipo *Object* con la palabra clave new
- Creando un *literal* del constructor *Object* con las llaves { }
- Definiendo un constructor con una estructura de propiedades y métodos y a partir de él crear nuevos objetos siguiendo su estructura.

Las dos primeras formas crean directamente una estructura de objeto con sus propios valores. Si lo comparamos con Java , las dos primeras formas estaríamos creando una "instancia" directamente de una Clase inexistente y a esta "instancia" le iríamos añadiendo propiedades y métodos (pero nunca tendrían definida una Clase con ésos métodos).

Siguiendo la comparación con Java, en la tercera forma de definir el objeto sí que estaríamos definiendo una Clase (Constructor), y podríamos entonces "instanciar" muchas veces ésa Clase.

Las dos primeras formas se utilizan sobre todo cuando solamente necesitamos guardar un conjunto de propiedades y valores como si fueran un array asociativo (aunque también le podemos asignar funciones).

A continuación vamos a ver las dos primeras formas para crear objetos.

3.4.1. Crear objetos directamente

Al crear un objeto directamente en JavaScript, serán estructuras de datos únicas que partiran de la clase Object a las que les podremos añadir propiedades y métodos en tiempo de ejecución sin tener que definir previamente una clase.

A continuación veremos algunas formas de definir nuestros propios objetos sin tener que crear primero la definición de su clase.

- A partir del constructor Object: asignando a una variable la palabra clave *new* seguido del constructor *Object*
- Como un literal: asignando a una variable las llaves *{ }.*

```
//creamos el objeto a partir del constructor Object
let infoPhone= new Object();

//creamos el objeto como un literal
let infoArbol = {};
```

En este ejemplo hemos almacenado un objeto vacío a una variable. Podemos añadir propiedades a un objeto simplemente escribiendo el nombre de la variable que contiene el objeto seguido de un punto, el nombre de la propiedad y asignándole el valor que queremos que tenga. Es decir, con la sintaxis:

nombreVariableObjeto.propiedad=valor;

En el siguiente ejemplo asignaremos a *infoPhone* la propiedad *marca* y *precio*.

```
let infoPhone= {};

infoPhone.marca ="LaHora"; //asignamos propiedades al objeto
infoPhone.precio=200;  //creamos el objeto
```

Podemos fácilmente ahora mostrar la propiedad *marca* de *infoPhone* accediendo a ella escribiendo el nombre del objeto seguido de un punto y el nombre de la propiedad. Por ejemplo vamos a mostrar ahora la marca y el precio por consola:

```
let infoPhone = new Object();

infoPhone.marca = "LaHora";
infoPhone.precio = 200;

console.log(infoPhone.marca); //mostrará “La Hora”
console.log(infoPhone.precio); //mostrará “200”
```

También podemos establecer y leer las propiedades de un objeto mediante la notación de corchetes [ ] indicando el nombre de la propiedad entre comillas como si fuera un array asociativo e incluso alternar una forma y la otra. De hecho lo objetos son utilizados comúnmente como un array asociativo. El siguiente código crea un objeto con información sobre un árbol alternando las dos formas vistas para asignar propiedades.

```
let infoArbol = {};

infoArbol["hoja"]="perenne";
infoArbol.edad=200;

console.log( infoArbol.hoja,":",infoArbol["edad"]); // “perenne:200”
```

Además, también podemos asignar funciones a un objeto gracias a que las funciones son tratadas en JavaScript como objetos. Siguiendo el ejemplo anterior hemos añadido *infoArborl* una función *muestraInfo()* y la hemos ejecutado:

```
let infoArbol = {};

infoArbol["hoja"] = "perenne";
infoArbol.edad = 200;

infoArbol.muestraInfo= function(){
   console.log(infoArbol.hoja, ":", infoArbol["edad"]); //perenne:200
}

infoArbol.muestraInfo();
```

Si nos fijamos en el ejemplo anterior, veremos que para acceder a las propiedades de *infoArbol* desde la función *muestaInfo* hemos hecho referencia al mismo objeto *infoArbol*.

Esto no muestra más problema porque al ser un literal, literalmente conocemos el nombre del objeto pero, ¿cómo podríamos hacer para referirnos desde dentro la función al propio objeto que contiene la función? Muy sencillo, solo necesitamos utilizar la palabra clave *this*.

En el siguiente ejemplo utilizamos la palabra clave *this* para acceder a las propiedades del propio objeto sin conocer su nombre:

```
let infoArbol = {};

infoArbol["hoja"] = "perenne";
infoArbol.edad = 200;

infoArbol.muestraInfo= function(){
   console.log(this.hoja, ":", this["edad"]); //perenne:200
}

infoArbol.muestraInfo();
```

3.4.2. Arrays asociativos

Las Arrays asociativas son aquellas que almacenan sus valores en posicions identificadas por un texto. En JavaScript podemos utilizar los objetos para “simular” arrays asociativas. En este caso se suele crear el objeto como un literal o crear un objeto array y añadirle propiedades posteriormente.

Podemos declarar un objeto como un literal con valores iniciales con la sintaxis:

let miArrayAssoc={ "clave1":"valor 1" , "clave2": 300 , ..etc..}

Si el valor de la propiedad es numérico no hemos de indicarlo entre comillas.

Como veremos más a delante esta sintaxis es la llamada notacion JSON.

En el siguiente ejemplo hemos creado una “array associativa”como un literal para almacenar la información de un arbol, luego le hemos añadido otra propiedad y posteriormente mostramos esa información por consola.

```
let infoArbol = {"hoja":"perenne" ,  "edad":200};
infoArbol["tipoMadera"]="dura";

console.log(this.hoja, ":", this["edad"]  , ":", this.tipoMadera); //perenne:200 :dura
```

3.4.2.1. **Eliminar una propiedad de un objeto**

Para eliminar una propiedad de un objeto hemos de utilizar la sentencia *delete* seguido de la propiedad que queremos eliminar. Por ejemplo: *delete miObjeto.laPropiedad*.

En el siguiente ejemplo se crea un arbol con dos propiedades y posteriormente se eliminan.

```
let infoArbol = {};
infoArbol["hoja"]="perenne";
infoArbol.edad=200;

delete infoArbol.edad;  // elimina edad:200
delete infoArbol.hoja;  // elimina hoja:perenne
```

3.4.2.2. **Recorrer las propiedades**

Para recorrer las propiedades de un objeto podemos utilizar la sintaxis*:*

for (let key in objeto){ }

Esta sintaxis recorrerá todos los nombres de las propiedades almacenados en el objeto y por cada clave se almacenará en la variable *key* y ejecutará el bloque de código. El resultado del siguiente código será:

hoja :perenne
edad : 200

```
let infoArbol = new Object();
infoArbol["hoja"]="perenne";
infoArbol.edad=200;

for(let key in infoArbol){ //recorremos las propiedades del objeto

   console.log(key+":"+infoArbol[key]); //hoja:perenne  edad:200

}
```

3.4.2.3. **Paso por referencia**

Hay que remarcar que en JavaScript (al igual que en Java) los objetos se pasan por referencia, no por valor. Es decir, que si a una variable le asignamos un objeto ya creado, no estaremos creando una copia, sino que estaremos definiendo una nueva variable que apunta al mismo objeto.

Si queremos crear efectivamente una copia deberemos crear un nuevo objeto y copiarle todas las propiedades. Pero hemos de tener presente que si una propiedad es un objeto, tendremos que crear esa propiedad de nuevo y copiarle todas las propiedades.

Una alternativa es utilizar el método *Object.assign()* pasando como parámetro el objeto a copiar. Pero Seguiremos teniendo el problema de aquellas propiedades que a su vez sean objetos.

El siguiente código muestra como el objeto *mismoArbol* se trata como una referencia a *infoArbol* y en cambio el objeto *otroArbol se* trata como una copia de *infoArbol*. El resultado de ejecutar el código será: *500=500=200*

```
let infoArbol = {};
infoArbol["hoja"] = "perenne";
infoArbol.edad = 200;

let mismoArbol = infoArbol; //referencia a infoArbol

let otroArbol = Object.assign({},infoArbol); //copia de infoArbol

infoArbol.edad=500;
console.log(infoArbol.edad+"="+mismoArbol.edad+"="+otroArbol.edad); //500=500=200
```

3.4.2.4. **Arrays con propiedades**

Una particularidad de JavaScript es que podemos añadir propiedades a cualquier objeto que hayamos creado. Esto también es aplicable para los arrays, con lo que en un mismo objeto vamos a poder almacenar valores en posiciones numéricas y en palabras clave.

En el siguiente ejemplo vemos como una vez declarado el array *infoArbol* le añadimos dos propiedades (hoja y edad) y luego seguimos añadiendo dos elementos en posiciones numéricas (0 y 2) dentro del array. Lo más curioso es que si miramos la longitd del array nos retornará 3, ya que el elemento situado en la posición numérica más alta del array está almacenado en la posición 3. Es decir, las propiedades de un array no se tienen en cuenta a la hora de contar el número de elementos del array.

```
let infoArbol = [];
infoArbol["hoja"] = "perenne"; //propiedad del array
infoArbol.edad = 200; //propiedad del array
infoArbol[0] = "rama"; //valor en la posición 0 del array
infoArbol[2] = "fruta"; //valor en la posición 2 del array
console.log(infoArbol.length); //3
console.log(infoArbol.hoja+"-"+infoArbol.["edad"]+"-"+infoArbol[0]+"-"+infoArbol[2]); //perenne-200-rama-fruta
```

Tal como vemos en el ejemplo anterior, el acceso a los distintos valores que contiene el array se hace de la misma forma a como hemos visto hasta ahora para los arrays y los objetos según si el valor es una propiedad del array o un valor del mismo.

Podemos recorrer los distintos valores y propiedades del array utilizado las iteraciones que hemos visto para recorrer los valores de un array y las propiedades de un objeto conociendo sus particularidades:

- Un bucle for que recorra la longitud del array accederá a todos los valores del array (también los undefined) y no accederá a las propiedades del array.
- Un bucle forEach recorrerá todas las claves numéricas del array que no contengan un valor undefined y no accederá a las propiedades del array.
- Un bucle for/in recorrerá todas las claves numéricas del array que no contengan un valor undefined y también todas las propiedades del array.

En el siguiente ejemplo podemos ver un uso de las tres formas para recorrer el array.

Con el bucle For imprime por consola los valores:

0:rama
1:undefined
2:fruta

Con el bucle forEach imprime por consola los valores:

0:rama
2:fruta

Con el bucle for/in imprime por consola los valores:

0:rama
2:fruta
hoja:perenne
edad:200

```
let infoArbol =[];
infoArbol["hoja"] = "perenne";
infoArbol.edad = 200;
infoArbol[0] = "rama";
infoArbol[2] = "fruta"
// bucle For para recorrer el array
for (let k = 0; k < infoArbol.length; k++) {
   console.log(k + ":" + infoArbol[k]); //0:rama, 1:undefined, 2:fruta
}

//bucle forEach para recorrer el array
infoArbol.forEach(function (atributo, key) {
   console.log(key + ":" + atributo);  //0:rama , 2:fruta
});

//for/in para recorrer los atributos del objeto
for (let key in infoArbol) {
   console.log(key + ":" + infoArbol[key]);
   //0:rama , 2:fruta , hoja:perenne,  edad:200
}
```

3.4.2.5. **JSON**

JavaScript Object Notation (JSON) es un formato de estructuración de datos vasado en texto que sigue la sintaxis de creación de objetos como literales de JavaScript con la excepción que las propiedades no pueden contener funciones. Las propiedades de un JSON solo pueden contener cadenas, números, arrays, booleanos y otros literales de objetos.

Como la notación JSON indica solo la forma de estructurar la información, puede ser utilizada independientemente de JavaScript. Actualmente es una notación muy utilizada gracias a su fácil comprensión, conversión a objeto y longitud compacta.

A continuación vamos a ver una notación JSON para estructurar información de un contacto:

```
{  "nombre":"Niki",
   "apellido":"Kol",
   "edad":35
}
```

```
//propiedad nombre con valor "Niki"
//propiedad apellido con valor "Kol"
// propiedad edad con valor 35
```

La notación JSON siempre empieza por "{" y termina con "}". Los distintos pares de propiedad y valor van separados entre ellos con comas "," y el nombre de la propiedad se separa del valor con dos puntos ":". En JSON es recomendable utilizar siempre comillas dobles para indicar el nombre de la propiedad y siempre utilizaremos comillas dobles para indicar que un valor es una string y escribiremos directamente los valores numéricos.

Para declarar un texto JSON en una variable JavaScript (en formato texto, no como un literal) resulta muy útil la utilización de los acentos abiertos " ` " para indicar un conjunto de texto sin tener en cuenta los saltos de línea.

Podemos estructurar también arrays utilizando su notación literal. Por ejemplo, a continuación declaramos una variable de tipo texto que contendrá una estructura JSON utilizando la notación literal para declarar un array y múltiples contactos. A demás hemos utilizado la notación literal para indicar que ahora el parámetro apellido contendrá un array con dos valores en ambos casos:

```
let arrayJSON = `[
  {
    "nombre": "Niki",
    "apellidos": ["Kol","Kel"],
    "edad": 35
  }
  , {
    "nombre": "Loki",
    "apellido": ["Moki","Doky"],
    "edad": 76
  }
]`;
```

A continuación veremos cómo podemos convertir de JSON a literales y viceversa.

3.4.2.5.1. Conversiones entre JSON y literales

Los JSON son solo textos con la notación de objetos literales JavaScript (pero textos al fin y al cabo), pero pueden ser fácilmente convertibles a objetos JavaScript (y viceversa) gracias al objeto global JSON.

Para convertir un texto con una estructura JSON a un literal solo es necesario utilizar el objeto global JSON y su método *parse()* con la sintaxis:

let nuevoLiteral = JSON.parse(textoJSON);

En el siguiente ejemplo convertimos la información del JSON a literal. A continuación accedemos y mostramos el nombre del primer contacto y finalmente almacenamos el segundo contacto en un objeto y mostramos su primer apellido:

```
let arrayJSON = `[
   {
      "nombre": "Niki",
      "apellido": ["Kol","Kel"],
      "edad": 35
   }
   , {
      "nombre": "Loki",
      "apellido": ["Moki","Doky"],
      "edad": 76
   }
]`;
let arrayContactos = JSON.parse(arrayJSON); //parseamos el JSON a literal
console.log(arrayContactos[0].nombre);  //extraemos el nombre del 1r contacto

let segundoContacto = arrayContactos[1]; //guardamos el 2o contacto en un objeto
console.log(segundoContacto.apellido[0]); //accedemos al 2o apellido
```

Para convertir un objeto o array a una estructura JSON solo es necesario utilizar el objeto global JSON y su método *stringify ()* con la sintaxis:

let textJSON = JSON.stringify (objetoJavaScript);

En el siguiente ejemplo convertimos un objeto JavaScript a texto con notación JSON y lo mostramos por consola. A continuación volvemos a transformar el texto a objeto JavaScript y accedemos y mostramos el título y el nombre del primer contacto:

```
let objetoContactos ={
   titulo:"mis contactos",
   contactos: [ //parametro contactos que contiene un array
      {
         "nombre": "Niki",
         "apellido": "Kol",
         "edad": 35
      }
      , {
         "nombre": "Loki",
         "apellido": "Moki",
         "edad": 76
      }
   ]
};
let textoJSON = JSON.stringify(objetoContactos); //obtenemos el JSON correspondiente con el objeto
console.log(textoJSON);
objetoContactos = JSON.parse(textoJSON);  //a partir del JSON obtenemos su objeto

console.log(objetoContactos.titulo); //mostramos la propiedad titulo
console.log(objetoContactos.contactos[0].nombre); //mostrams el nombre de 1r contacto
```

La sentencia *console.log(textoJSON)* mostrará por pantalla el texto:

{"titulo":"mis contactos","contactos":[{"nombre":"Niki","apellido":"Kol","edad":35}, {"nombre":"Loki","apellido":"Moki","edad":76}]}

3.5. Clases

En JavaScript existen diversas formas distintas para definir una Clase con sus propiedades y métodos, así que nos centraremos en las dos formas más amigables y comprensibles:

- Usando una función constructora
- Usando la palabra reservada class

3.5.1. Función constructora

Para definir un constructor definiendo una función constructora simplemente tenemos que definir la función que vamos a utilizar como constructora como si fuera la definición de una clase. Una vez definida ya podremos crear un objeto utilizando la función como *constructor*.

Cuando en JavaScript hablemos de "función constructora" o "constructor" nos estaremos refiriendo a la estructura que utilizaremos para inicializar una variable (excepto cuando estemos trabajando con Class, en donde "constructor" adquiere el mismo significado que en Java)

En el siguiente ejemplo hemos creado una función constructora de nombre "ListaCompra" (con la intención de pueda estructura información sobre una lista de la compra) y hemos creado un nuevo objeto de nombre "miLista" y del tipo "ListaCompra".

```
function ListaCompra(){
}
let miLista = new ListaCompra();
```

Pero claro, no tiene mucho sentido definir una función constructora si no le indicamos una estructura formada por propiedades y/o métodos. Cuando queramos vincular a una función constructora una propiedad solo necesitamos declarar la propiedad dentro de la función como si fuera una variable pero utilizando la palabra reservada *this.* en vez de *let* o *var*.

En el siguiente ejemplo hemos añadido al constructor "ListaCompra" una propiedad "propietario" con un valor por defecto "Gin" y una propiedad "items" con una array vacía. Después hemos creado dos variables del tipo "ListaCompra" y modificado en una de ellas su propiedad "propietario" para comprobar que son objetos que no comparten valores, solo la estructura base.

```
function ListaCompra(){
    this.propietario="Gin";
    this.items=[];
}
let miLista = new ListaCompra();
let tuLista = new ListaCompra();
tuLista.propietario="Jon";

console.log(miLista.propietario); //Gin
console.log(tuLista.propietario);//Jon
```

Para convertir una propiedad en un método solo necesitamos asignar a la propiedad correspondiente una función. Muy importante es que siempre que desde el mismo constructor queramos hacer referencia a una propiedad suya, deberemos añadir el prefijo "this." delante del nombre de la propiedad.

En el siguiente ejemplo añadimos una propiedad "this.saludoPersonal" que se transforma en método al asignarle una función encargada de mostrar un saludo por pantalla.

```
function ListaCompra(){
  this.propietario="Gin";
  this.items=[];
  this.saludoPersonal=function(){
    console.log("Hola "+this.propietario); //importantísimo el this
  }
}
let miLista = new ListaCompra();
let tuLista = new ListaCompra();
tuLista.propietario="Jon";

miLista.saludoPersonal(); //Hola Gin
tuLista.saludoPersonal();//Hola Jon
```

Si ahora nos preguntamos si existe algo parecido a un "constructor" típico de Java, entendido como una función que se ejecuta al crear un objeto a partir de una "Clase", la respuesta es que la función constructora es en sí misma un "constructor" (por ello se llama función constructora). Es decir, cuando en nuestro código de ejemplo anterior se ha ejecutado la sentencia "new ListaCompra();", antes de retornar la estructura definida en "ListaCompra" se ha evaluado la función "ListaCompra".

En el siguiente ejemplo vamos a aprovechar esto para pasar directamente el nombre del propietario a “ListaCompra” y así aprovechar la función constructora para establecer ese valor como el propietario:

```
function ListaCompra(propietario="") {   //por defecto ""
   this.propietario = propietario; //el propietario será el valor pasado por parámetro
   this.items = [];
   this.saludoPersonal = function () {
      console.log("Hola " + this.propietario);
   }
}
let miLista = new ListaCompra("Gin"); //pasamos el valor Gin al constructor
let tuLista = new ListaCompra("Jon");  //pasamos el valor Jon al constructor

miLista.saludoPersonal(); //Hola Gin
tuLista.saludoPersonal(); //Hola Jon
```

3.5.2. Definir una Class

Otra forma de definir una clase es utilizando la palabra reservada “class” seguido del nombre de la clase y las llaves “{ }”. En el siguiente ejemplo hemos creado una clase de nombre “ListaCompra” y unnuevo objeto de nombre “miLista” y del tipo “ListaCompra”.

```
class ListaCompra{
}
let miLista = new ListaCompra();
```

Podemos añadir vincular a la clase una propiedad declarando directamente la propiedad (sin utilizar las palabras reservadas *this.*, *let* o *var)*.

En el siguiente ejemplo hemos añadido a la clase “ListaCompra” una propiedad “propietario” con un valor por defecto “Gin” y una propiedad “items” con una array vacía. Después hemos creado dos variables del tipo “ListaCompra” y modificado en una de ellas su propiedad “propietario” para comprobar que son objetos que no comparten valores, solo la estructura base.

```
class ListaCompra {
   propietario = "Gin"; // con class no necesitamos el this
   items = [];
}

let miLista = new ListaCompra();
let tuLista = new ListaCompra();
tuLista.propietario="Jon";

console.log(miLista.propietario); //Gin
console.log(tuLista.propietario); //Jon
```

Para convertir una propiedad en un método solo necesitamos asignar a la propiedad correspondiente una función. Muy importante es que siempre que desde el mismo constructor queramos hacer referencia a una propiedad suya, deberemos añadir el prefijo "this." delante del nombre de la propiedad.

En el siguiente ejemplo añadimos el método "saludoPersonal" asignándole una función encargada de mostrar un saludo por pantalla.

```
class ListaCompra {
   propietario = "Gin"; // con class no necesitamos el this
   items = [];
   saludoPersonal = function () { // con class no necesitamos el this
      console.log("Hola " + this.propietario); //aquí si necesitamos el this
   }
}

let miLista = new ListaCompra();
let tuLista = new ListaCompra();
tuLista.propietario="Jon";
miLista.saludoPersonal(); //Hola Gin
tuLista.saludoPersonal();//Hola Jon
```

Si queremos definir una función para que sirva como "constructor", es decir, para que se ejecute al crear un objeto a partir de la "Clase", basta con definir un método con el nombre "construtor". Solo puede haber un método con el nombre "constructor" dentro de una misma clase.

En el siguiente ejemplo vamos a aprovechar esto para pasar directamente el nombre del propietario a “ListaCompra” y así aprovechar la función constructora para establecer ese valor como el propietario:

```
class ListaCompra {
   propietario = "";
   items = [];
   saludoPersonal = function () {
      console.log("Hola " + this.propietario);
   }
   constructor(propietario=""){ //función que se ejecuta al crear un objeto de ésta clase
      this.propietario=propietario;
   }
}
let miLista = new ListaCompra("Gin"); //pasamos el valor Gin al constructor
let tuLista = new ListaCompra("Jon");  //pasamos el valor Jon al constructor

miLista.saludoPersonal(); //Hola Gin
tuLista.saludoPersonal(); //Hola Jon
```

En el siguiente video podemos ver como crear una clase y objetos y depurar errores derivados en JavaScript.

Video: Crear y utilizar clases

Recursos y enlaces

- Codepen editor online de JavaScript

- JSFiddle editor colaborativo online de JavaScript

- Información en W3Schools sobre arrays

- Información en MDN sobre clases

Conceptos clave

- **Literal**: variable que ha sido creada inicializándola directamente sin un constructor.
- **this**: hace referencia al propio objeto que contiene el método o propiedad.
- **Constructor**: función que se ejecuta al crear una nueva instancia de una variable.
- **Función anónima:** función que se define sin un nombre.

Test de autoevaluación

¿Qué tipo de bucle recorre todas las claves numéricas de un array y sus propiedades?

a) for/in
b) for
c) forLoop
d) forEach

¿Qué tipo de bucle recorre todas las claves numéricas de un array aunque tengan un valor *undefined*?

a) for/in
b) for
c) forLoop
d) forEach

¿Qué palabra clave usaremos para acceder a las propiedades de un objeto desde un método del mismo objeto?

a) self
b) super
c) then
d) this

Ponlo en práctica

Actividad 1

Crea un array de con 3 valores numéricos. Programa la aplicación web con un 3 botones para que al clicar el 1o muestre todos los valores del array en el HTML, al clicar al 2o pemita añadir un valor introducdo en un input, al clicar en el 3o elimine el 1r valordel array.

Actividad 2

Crea un array asociativo que almacene 2 matrículas y para cada una la marca y kilometraje de un coche. Programa la aplicación web con un 3 botones para que al clicar el 1o muestre todos los valores del JSON en el HTML, al clicar al 2o pemita añadir una matrícula, al clicar al 3o elimine la matrícula que escriba el usuario.

Actividad 3

Crea una Clase que estructure información sobre un bocata almacenando el tipo de pan, ingrediente principal, porcentaje de bocata sin comer y una función de nombre "comer" que al ejecutarse reste 10 al porcentaje de bocata sin comer (sin poder llegar a un número negativo). Programa la aplicación web con dos bocatas con distintos ingredientes y tipo de pan. Para cada bocata añade un boton para mostrar la información del bocata y otro que llame a la función comer del bocata correspondiente.

SOLUCIONARIOS

Test de autoevaluación

¿Qué tipo de bucle recorre todas las claves numéricas de un array y sus propiedades?

e) for/in

f) for

g) forLoop

h) forEach

¿Qué tipo de bucle recorre todas las claves numéricas de un array aunque tengan un valor *undefined*?

e) for/in

f) for

g) forLoop

h) forEach

¿Qué palabra clave usaremos para acceder a las propiedades de un objeto desde un método del mismo objeto?

e) self

f) super

g) then

h) this

Ponlo en práctica

Actividad 1

Crea un array de con 3 valores numéricos. Programa la aplicación web con un 3 botones para que al clicar el 1o muestre todos los valores del array en el HTML , al clicar al 2o pemita añadir un valor introducdo en un input , al clicar en el 3o elimine el 1r valordel array.

Los solucionarios están disponibles en la versión interactiva del aula.

Actividad 2

Crea un array asociativo que almacene 2 matrículas y para cada una la marca y kilometraje de un coche. Programa la aplicación web con un 3 botones para que al clicar el 1o muestre todos los valores del JSON en el HTML, al clicar al 2o pemita añadir una matrícula, al clicar al 3o elimine la matricula que escriba el usuario.

Los solucionarios están disponibles en la versión interactiva del aula.

Actividad 3

Crea una Clase que estructure información sobre un bocata almacenando el tipo de pan, ingrediente principal , porcentaje de bocata sin comer y una función de nombre "comer" que al ejecutarse reste 10 al porcentaje de bocata sin comer (sin poder llegar a un número negativo). Programa la aplicación web con dos bocatas con distintos ingredientes y tipo de pan. Para cada bocata añade un boton para mostrar la información del bocata y otro que llame a la función comer del bocata correspondiente.

Los solucionarios están disponibles en la versión interactiva del aula.