M4C8 – Course Checkpoint 8

- ¿Qué tipo de bucles hay en JS?
- ¿Cuáles son las diferencias entre const, let y var?
- ¿Qué es una función de flecha?
- ¿Qué es la deconstrucción variable?
- ¿Qué hace el operador de spread JS?
- ¿Qué es OOP?
- ¿Qué es una promesa JS?
- ¿Qué hacen async y await por nosotros?

¿Qué tipo de bucles hay en JS?

Los bucles permiten repetir un bloque de código varias veces. Los bucles son útiles para poder llevar a cabo repeticiones de código necesarias para poder hacer funcionar correctamente las funciones.

Gracias al uso de bucles se puede evitar duplicar código, gracias a ello se crea un código más limpio y fácil de interpretar, además de ser más ligero en cuanto al rendimiento

Aunque también se debe tener en cuenta que dentro de lo posible hay que evitar crear bucles son excesivas repeticiones ya que aunque el código se reutilice dentro del bucle, a nivel de eficiencia, tener excesivas repeticiones puede ser costoso.

Existen varios tipos de bucles.

- Bucle 'For'
  - Permite repetir un bloque de código un número determinado de veces.
  - Ejemplo:

```
for (let myNumber = 0; myNumber < 5; myNumber++) {
  console.log(myNumber);
}
```

    - Devuelve los números del 0 al 4 (Incluidos)

  - Análisis del código:
    - Una vez escrito 'for' se debe añadir dentro de los paréntesis los condicionales y se puede dividir en tres partes:
      - La primera en nuestro ejemplo es 'let myNumber = 0', esta es la variable y dato inicial en el que se va a basar para hacer el bucle. La variable en cuestión puede ser creada dentro de los paréntesis o fuera del bucle y después enlazándolo dentro del paréntesis añadiendo únicamente el nombre de la variable.
      - La segunda parte es 'myNumber > 5', esta es la condición que se debe cumplir para que el bucle continúe repitiéndose, en este caso nos dice que mientras la variable 'myNumber' sea menor que 5, entonces el bucle se continuara repitiendo.
      - La ultima parte es 'myNumber++', esto hace que por cada una de las repeticiones se sume una unidad a la variable 'myNumber' por lo que, y usando nuestro ejemplo, con cada repetición el valor de 'myNumber' aumentara en una unidad y cuando llegue a 5, el bucle se detendrá.
      - Para separar cada una de las tres partes se debe usar el punto y coma ';'

- Bucle 'for in'
  - Permite recorrer todos los datos dentro de un objeto
  - Ejemplo:

```
const precio = {
  manzana: 1,
  naranja: 2,
  pera: 3
};

for (let valor in precio) {
  console.log(valor  + ' - ' + precio[valor]);
}
```

    - Devuelve:

```
Console
"manzana - 1"
"naranja - 2"
"pera - 3"
```

- Análisis del código:
  - La línea 1 es la creación de una variable tipo objeto con tres elementos.
  - En la línea 7, después del 'for' se debe poner entre paréntesis los datos del bucle y se puede dividir en dos partes:
    - La primera parte es la creación de una variable usando 'let' que servirá para individualizar cada uno de los elementos dentro del propio bucle. El nombre de esta variable es libre, pero se recomienda que sea intuitivo y fácil de interpretar.
    - La segunda parte es el nombre de la variable con la que se va a trabajar, en nuestro ejemplo es 'precio'
    - Ambas partes están separadas por 'in'
  - La explicación de lo que hay dentro del paréntesis es que el bucle va a coger la variable externa 'precio' y hará una repetición por cada uno de sus elementos dándole el nombre de variable interna'valor'

- Bucle 'While'
  - Permite repetir un bloque de código siempre que se cumpla una condición
  - Ejemplo:

```
1  let myNumber = 0;
2  while (myNumber < 5) {
3    console.log(myNumber);
4    myNumber++;
5  }
```

  - Devuelve los números del 0 al 4 (Ambos incluidos)

  - Análisis del código:
    - La línea 1 es la creación de una variable externa que se usara para crear la condición del bucle.
    - En la línea 2, debemos escribir While y después entre paréntesis la condición que se debe cumplir para que el bucle continúe repitiéndose, en nuestro ejemplo el bucle se repetirá siempre que la variable 'myNumber' sea menor que 5. Una vez que la variable sea 5 o mayor, entonces el bucle se detendrá.
    - Dentro del bucle, las líneas 2.1 y 2.2 lo que hacen es sacar en consola el valor en tiempo real de la variable 'myNumber' y después sumarle una unidad a la variable 'myNumber' para que después de 5 repeticiones el bucle detecte que la variable 'myNumber' es igual o mayor que 5 y se detenga.

- Bucle 'Do While'
  - Es similar al bucle 'While' con la diferencia de que en este caso ejecuta el código de forma forzada al menos una vez, se cumpla o no la condición.
  - Ejemplo 1 – La condición se cumple varias veces

```
1  let myNumber = 0;
2  do {
3    console.log(myNumber);
4    2.2.  myNumber++;
5  } while (myNumber < 5);
```

  - Devuelve los números del 0 al 4 (Ambos inluidos)

- Ejemplo 2 – La condición no se cumple

```
let myNumber = 6;
do {
  console.log(myNumber);
  myNumber++;
} while (myNumber < 5);
```

  - Devuelve 6

- Análisis del código:
  - En la línea 1 se crea la variable ‘myNumber’ que se usara para la condición del bucle ‘while’
  - En la línea 2 se crea usando ‘do’ el código que se repetirá con el bucle, hay que tener en cuenta que independientemente con que si se cumple la condición, este código del bucle se va a repetir al menos una única vez
  - Las líneas 2.1 y 2.2 son para sacar a tiempo real en consola el valor de la variable ‘myNumber’ y sumarle una unidad a la variable ‘myNumber’
  - La línea 3 es el cierre de ‘do’ y la creación del ‘while’ con su condición dentro del paréntesis, que en nuestro ejemplo es que se repetirá siempre que el valor de ‘myNumber’ sea más pequeño que 5.

Es muy importante tener en cuenta cuando trabajamos con bucles que debemos tener claro en qué momento el bucle llega a su fin y de esta forma evitar que se haga un bucle infinito que pueda saturar y dañar el sistema completo.

En bucles como el ‘for in’ es más fácil ya que depende del número de elementos que existen en la variable externa asociada, con este tipo de bucle podemos tener la seguridad de que una vez que ha pasado por todos los elementos de la variable, el bucle se detendrá.

Pero en bucles como el ‘while’ no depende del número de elementos de una variable externa, sino de las condiciones que le hemos informado, por lo que si las condiciones no están bien hechas o no le damos una salida independiente, si que podremos tener un bucle infinito que bloqueara todo el sistema.

Algo a tener en cuenta es evitar modificar dentro del bucle la variable usada para la condición de repetición, si no se tiene totalmente claro lo que se está modificando podría tener consecuencias en las repeticiones y podría crear un bucle infinito o una detención inesperada.

Si hay riesgo de un bucle infinito o se necesita detener un bucle sin tener en cuenta la condición, se usa ‘break’ y de esta forma automáticamente el bucle se detiene.

Es posible poder anidar bucles dentro de bucles y también crear variables y diferente tipo de código que se usara internamente para la repetición del bucle

Fuentes:

- DEV Community
- Developer Mozilla
- Developer Mozilla
- Lenguaje JS
- Developer Mozilla (Break)

¿Cuáles son las diferencias entre const, let y var?

- var
  - Es la forma tradicional de declarar variables en JavaScript.
  - Las variables var están dentro del hoisting, lo que significa que pueden ser accedidas antes de ser declaradas en el código.
  - Las variables var tienen un alcance global o local dependiendo de donde se declaren.
  - Las variables con var pueden ser redeclaradas de nuevo.
  - Ejemplos
    - Crear variable
      - var myName = 'Ivan';
    - Sobrescribir variable
      - var myName = 'Ivan';
      - myName = 'Ivan Carballo Machuca';
    - Redeclarar variable:
      - var myName = 'Ivan';
      - var myName = 'Ivan Carballo';

- let
  - Se introdujo como una alternativa a var.
  - Las variables declaradas con let tienen un alcance de bloque, lo que significa que solo pueden ser accedidas dentro del bloque donde se declaran.
  - Las variables con let no están dentro de hoisting, por lo que deben ser declaradas antes de ser utilizadas en el código.
  - Las variables con let se pueden sobrescribir pero no redeclarar usando let.
  - Ejemplos
    - Crear variable
      - let myName = 'Ivan';
    - Sobrescribir variable
      - let myName = 'Ivan';
      - myName = 'Ivan Carballo';
    - Redeclarar variable
      - Let myName = 'Ivan';
      - Let myName = 'Ivan Carballo';
      - Esta redeclaracion usando de nuevo let dará error

- const
  - Se utiliza para declarar variables cuyo valor no cambiará a lo largo de la ejecución del programa.
  - Las variables con const no pueden ser reasignadas y deben ser inicializadas al momento de la declaración.
  - Al igual que let, las variables con const tienen un alcance de bloque y no están sujetas al hoisting.
  - Ejemplos
    - Crear variable
      - const myName = 'Ivan';
    - Sobrescribir variable
      - const myName = 'Ivan';
      - myName = 'Ivan Carballo';
      - Esta sobre escritura dará error
    - Redeclarar variable
      - const myName = 'Ivan';
      - const myName = 'Ivan Carballo';
      - Esta redeclaracion dará error

Las diferencias entre const, let y var radican en su alcance, su comportamiento respecto al hoisting y la posibilidad de reasignar valores.

var es la variable que más se adapta, pero también la que más errores puede provocar al permitir modificaciones sin ningún tipo de control de nombres.

let es una palabra clave más actual que mejora la seguridad de var en cuanto a conflictos de nombre y que da un uso dentro de bloque.

const se usa para declarar variables que no deben modificarse y deben ser estáticas a lo largo de todo el código.

Fuentes:

- Medium
- Free Code Camp

¿Qué es una función de flecha?

Una función de flecha es una forma de definir funciones de forma más rápida y limpia que con la forma tradicional.

Las funciones de flecha no tienen su propio contexto this, lo que significa que heredan el this de la función que las contiene. Esto las hace útiles en ciertos escenarios, como en funciones de devolución de llamada, funciones de orden superior y funciones de mapa, filtro y reducción en arrays.

- Comparativa entre una función tradicional y una función de flecha:
  - Función tradicional

```
function contador(numero) {
  return numero + 100;
}
```

  - Función de flecha

```
let contador = (numero) => numero + 100;
```

  - Análisis del código:
    - La apertura ‘function’ desaparece y es sustituida por la declaración ‘let’ que usa el nombre de la función como nombre de la variable y su valor son los argumentos junto con el bloque de código
    - Después de los argumentos debemos escribir ‘=>’ para indicar que a partir de ese punto es cuando da comienzo el bloque de código de la función.
    - En el caso de que el bloque de código sea únicamente un ‘return’, se puedes eliminar las llaves ‘{}’, la palabra ‘return’ y terminar con el punto y coma ‘;’. Cuando únicamente hay una línea, el sistema sabe que debe ser un return por lo que se puede ahorrar tener que informarlo.
    - En el caso contrario, si además del return hay mas código, entonces sí que se deben respetar las llaves ‘{}’ y también mantener la palabra ‘return’.

- También es válido para funciones que no tienen un nombre de función asignado:
  - Función tradicional

```
function (numero) {
  return numero + 100;
}
```

  - Función de flecha

```
(numero) => numero + 100;
```

  - Análisis del código:
    - Usa la misma dinámica que la función con un nombre asignado, salvo que al no tener ningún nombre, la parte de ‘let contador =’ no es necesaria ya que no existe en la función con estructura tradicional.

- También es válido para funciones a las que no se les pasa argumentos:
  - Función tradicional

```
function () {
  return numero + 100;
}
```

  - Función de flecha

```
() => numero + 100;
```

  - Análisis de código:
    - Cuando no se le da ningún argumento a la función, lo que se debe hacer es dejar el paréntesis vacio tal como está en la función con la sintaxis tradicional, el resto es todo de la misma forma que en el resto de ejemplos.

Hay algunos puntos que debo matizar para un correcto uso de las funciones de flechas.

Las funciones de flecha son anónimas, por esa razón cuando la función tradicional tiene un nombre asignado, pasamos ese nombre como valor de una variable y de esta forma se puede seguir llamando a la función desde otro lugar del código.

Cuando no existen argumentos, realmente no es necesario poner los paréntesis vacios, se pueden eliminar, pero para poder mejorar la lectura y compresión es recomendable dejarlos vacios.

Cuando únicamente hay un solo argumento, los paréntesis son opcionales, pero si hay dos o más argumentos el uso del paréntesis es obligatorio.

Cuando en una función tenemos únicamente el return, el uso de las llaves '{}' y la palabra 'return' son opcionales, pero no es posible usar uno sí y otro no, o se usan los dos o no se usa ninguno, pero no es posible usar las llaves '{}' sin usar 'return' y viceversa, no es posible usar 'return' y no las llaves '{}'.

Fuentes:

- Free Code Camp
- Developer Mozilla
- dCreations
- Desarrollo Web

¿Qué es la deconstrucción variable?

Es una característica que permite extraer valores de un objeto o array para poder asignarlo a una nueva variable individual, de forma sencilla y más fácil de entender e interpretar.

Usando esta característica se puede ahorrar líneas de código y hacer más intuitivo el código.

- Ejemplos usando arrays
  - Si se quisiera extraer datos de un array de la forma tradicional seria de la siguiente forma:

```
const dato = ['dato primero', 'dato segundo', 'dato tercero', 'dato cuarto'];
let dato1 = dato[0];
let dato4 = dato[3];
console.log(dato1 + ' - ' + dato4);
```

  - Usando la desconstrucción quedaría de la siguiente manera:

```
const dato = ['dato primero', 'dato segundo', 'dato tercero', 'dato cuarto'];
let [dato1, , ,dato4] = dato;
console.log(dato1 + ' - ' + dato4);
```

    - En ambos casos devuelve ‘dato primero – dato cuarto’

- Ejemplos usando objetos
  - Si se quisiera extraer datos de un array de la forma tradicional seria de la siguiente forma:

```
const dato = {dato1:'dato primero', dato2:'dato segundo', dato3:'dato tercero'};
let dato1 = dato['dato1'];
let dato3 = dato['dato3'];
console.log(dato1 + ' - ' + dato3);
```

  - Usando la desconstrucción quedaría de la siguiente manera:

```
const dato = {dato1:'dato primero', dato2:'dato segundo', dato3:'dato tercero'};
let {dato1,dato3} = dato; //opción 1
console.log(dato1 + ' - ' + dato3);

let {dato1:primero,dato3:tercero} = dato; //opción 2
console.log(primero + ' - ' + tercero);
```

    - Con ambas opciones devuelve ‘dato primero-dato tercero’

- Análisis del código
  - Tanto con arrays como con objetos la dinámica es la misma, usando una única línea poder crear tantas variables como datos se necesitan, en vez de tener que crear variables independientes.
  - En este ejemplo puede parecer poco, pero si es un array u objeto grande y además se tiene que hacer con diferentes arrays u objetos a lo largo del código entonces el número de líneas creando variables puede llegar a ser muy grande.
  - En el ejemplo del array tenemos la siguiente línea ‘let [dato1, , ,dato4] = dato;’
    - Comenzamos declarando la variable con let seguido de unos corchetes ‘[]’. Dentro de los corchetes debemos informar de que datos queremos sacar del array y que nombre de variable darle.
    - En nuestro caso queremos el primer y cuarto dato, por lo que le damos el nombre de variable ‘dato1’ y ‘dato4’
    - Para especificar que solo queremos esos dos y no el resto, usamos las comas dejando espacios vacios a donde correspondería el nombre de las variables de los otros dos elementos.
    - Por último y después de los corchetes, debemos poner un igual ‘=’ y el nombre del array al que hacemos referencia.
    - Con esta línea lo que hemos logrado es tener dos variables ‘let’ llamadas ‘dato1’ y ‘dato4’ cuyo valor es ‘dato primero’ y ‘dato cuarto’ respectivamente.

- En el ejemplo del objeto tenemos dos opciones diferentes para su uso
    - La dinámica es aun más sencilla que usando un array, en este caso también debemos comenzar con una declaración de variable, en el ejemplo usamos ‘let’
    - Después de la declaración debemos abrir llaves ‘{}’ ya que en este caso vamos a enlazar con un objeto
    - Aquí es donde detallare las dos opciones posibles, comenzare con la opción 1
        - Tenemos la siguiente línea ‘let {dato1,dato3} = dato;’
        - Al estar enlazando con un objeto únicamente debemos informar de las llaves que identifican los elementos y se crearan como nombre de variables ‘let’
        - Es decir, en nuestro caso tenemos dos variables ‘let’ con nombre ‘dato1’ y ‘dato3’ cuyos valores son ‘dato primero’ y ‘dato tercero’ respectivamente.
    - Aquí detallare la opción 2
        - Tenemos la siguiente línea ‘let {dato1:primero,dato3:tercero} = dato;’
        - Si nuestra idea es crear las variables pero poder nosotros asignarles nombres en concreto lo podemos hacer tal como se ve en nuestro ejemplo, a cada una de las llaves se le está asignando un nombre que será el que tendrá nuestras dos nuevas variables.
        - Es decir, en este ejemplo estamos trayendo el valor asociado a las llaves ‘dato1’ y ‘dato3’ y en vez de usar ese mismo nombre de llaves como nombre de variable, estamos creando las variables con nombre ‘primero’ y ‘tercero’ y es a estas variables a las que estamos asignando los valores ‘dato primero’ y ‘dato tercero’ respectivamente.
    - Por último y después de cerrar las llaves ‘{}’ ponemos el símbolo igual ‘=’ y el nombre del objeto al que hacemos referencia.

Fuentes:

- Developer Mozilla
- Free Code Camp
- Carlos Escorche Blog

¿Qué hace el operador de spread JS?

El operador de spread es representado por tres puntos '...'. Este operador permite expandir elementos de un array o un objeto donde se necesitan múltiples elementos y poder trabajar con ellos. Esto hace que el código sea más fácil y limpio.

Este operador se utiliza para combinar arrays u objetos de una manera más directa y legible. También se puede utilizar para descomponer en elementos individuales y después poder trabajar con ellos.

También es posible usarlo para definir argumentos a una función sin necesidad de escribir cada elemento de forma individual.

- Ejemplos de uso
  - Combinar dos arrays:

```
const myArray1 = ['dato1', 'dato2', 'dato3'];
const myArray2 = ['dato4', 'dato5', 'dato6'];
let myArray3 = [...myArray1, ...myArray2];
```

    - Devuelve `["dato1","dato2","dato3","dato4","dato5","dato6"]`

  - Combinar dos objetos:

```
const myObjeto1 = {dato1:'primero', dato2:'segundo'};
const myObjeto2 = {dato3:'tercero', dato4:'cuarto'};
let myObjeto3 = {...myObjeto1, ...myObjeto2};
```

    - Devuelve

```
{
  "dato1": "primero",
  "dato2": "segundo",
  "dato3": "tercero",
  "dato4": "cuarto"
}
```

  - Uso en los argumentos de una función:

```
const argumentos = ['arg1', 'arg2', 'arg3', 'arg4'];
function myFuncion(...argumentos) {};
```

    - Los cuatro argumentos se han pasado a la función sin necesidad de escribir los cuatro dentro de los paréntesis manualmente.

  - Añadir nuevo elemento a un array:

```
const myArray = ['dato1', 'dato2', 'dato3'];
let newArray = [...myArray, 'dato4'];
```

    - Devuelve `["dato1","dato2","dato3","dato4"]`
    - Cambiando el orden de los elementos de la variable 'newArray' se puede hacer que los elementos nuevos se añadan al principio o al final.

  - Añadir nuevo elemento a un objeto

```
const myObjeto = {dato1:'primero', dato2:'segundo', dato3:'tercero'};
let newObjeto = {...myObjeto, dato4:'cuarto'};
```

    - Devuelve

```
{
  "dato1": "primero",
  "dato2": "segundo",
  "dato3": "tercero",
  "dato4": "cuarto"
}
```

Tal como se ve en los ejemplos, la forma de usar el Spread es muy sencilla, basta con poner los tres puntos '…' antes del objeto o array para que se puedan usar los elementos por separado.

Hay que tener en cuenta que si se usa con un array debemos usar corchetes '[]' y si se usa con un objeto son llaves '{}'
Es posible usar spread junto con la deconstrucción para poder hacer extracciones más concretas.

- Ejemplo de uso

```
const myArray = ['dato1', 'dato2', 'dato3', 'dato4', 'dato5', 'dato6'];
let [primero, segundo, ...restante] = myArray;
```

  - Devoluciones:
    - La variable 'primero' devolverá 'dato1'
    - La variable 'segundo' devolverá 'dato2'
    - La variable 'restante' devolverá ['dato3', 'dato4', 'dato5', 'dato6']

De esta forma se puede crear un array con todos los elementos que no hemos individualizado para poder trabajar con ellos en otro punto del código sin necesidad de tener que usar de nuevo la variable 'myArray'

Fuentes:

- Developer Mozilla
- Alex Tomas Blog
- Kinsta

¿Qué es OOP?

La programación orientada a objetos (OOP) es un modelo de programación en el que el diseño se organiza alrededor de datos u objetos, en vez de usar funciones. Se enfoca en los objetos que se necesitan manipular, en lugar de centrarse en la lógica necesaria para esa manipulación. Un objeto se puede definir como un grupo de datos con atributos y comportamientos únicos.

La principal característica de este tipo de programación es que soporta objetos que tienen un tipo o clase asociado. Esas clases pueden heredar atributos de una clase superior. Por ello este enfoque de programación se utiliza en programas grandes y complejos que se deben actualizar de forma habitual.

Cuando se diseñan las clases se pueden usar en distintas partes del código. La técnica de herencia ahorra tiempo porque permite crear una clase genérica y luego definir las subclases que heredan sus rasgos, por lo que te ahorras tener que reescribir esas funciones de nuevo.

Permite gestionar objetos y funciones fácilmente para actualizar el código y se logra un ahorro de tiempo y líneas de código por su mayor facilidad.

Algunas de las ventajas son las siguientes:

- Abstracción - Permite crear objetos del mundo real de forma más fiel en el código facilitando su comprensión y modificación.
- Encapsulamiento - Permite ocultar la complejidad interna de los objetos y mostrar solo los datos necesarios para interactuar con ellos, lo que mejora la seguridad y el mantenimiento del código.
- Herencia - Permite la creación de nuevas clases basadas en clases ya creadas anteriormente, lo que facilita la reutilización de código.
- Polimorfismo - Permite que objetos del mismo tipo respondan de manera diferente a una misma acción, lo que mejora la flexibilidad y la extensibilidad del código.

- Ejemplo de definición de una clase con constructor

```
class Coches {
  constructor(marca, modelo) {
    this.marca = marca;
    this.modelo = modelo;
  }

  presentacion() {
    console.log(`El ${this.modelo} es el nuevo modelo de la marca ${this.marca}.`);
  }
}

let miCoche = new Coches('Citroen', 'DS4');
miCoche.presentacion();
```

- Análisis del código:
  - Tenemos una clase llamada ‘Coches’ y se añade un constructor que recibe los parámetros ‘marca’ y ‘modelo’, y asigna estos valores a las propiedades ‘marca’ y ‘modelo’ del objeto usando ‘this’
  - La clase tiene un método llamado ‘presentación’ que devuelve un mensaje con el modelo y la marca del coche que se enlaza desde el constructor usando ‘this’.
  - Se crea una instancia llamada ‘miCoche’ que llama a la clase ‘Coches’ y le da los argumentos para ‘marca’ y ‘modelo’
  - Por último se llama al método ‘presentación’ de la instancia ‘miCoche’ para que nos devuelva un mensaje usando los argumentos que hemos pasado desde la instancia ‘miCoche’.
  - El mensaje que devuelve es: El DS4 es el nuevo modelo de la marca Citroen.

- Ejemplo del uso de herencias usando el ejemplo anterior

```
class Coche {
  constructor(marca, modelo) {
    this.marca = marca;
    this.modelo = modelo;
  }

  presentacion() {
    console.log(`El ${this.modelo} es el nuevo modelo de la marca ${this.marca}`);
  }
}

class Motorizacion extends Coche {
  novedad() {
    console.log(`El ${this.marca} ${this.modelo} usa motor fabricado por ${this.marca}`);
  }
}

const miMotor = new Motorizacion('Citroen', 'DS4');

miMotor.presentacion();
miMotor.novedad();
```

- Análisis del código:
  - Partimos del mismo código que en el ejemplo anterior, tenemos una clase llamada ‘Coche’ con un constructor que trabaja con los argumentos ‘marca’ y ‘modelo’ y un método llamado ‘presentación’ que devuelve un mensaje con los argumentos usando ‘this’
  - Creamos una nueva clase llamada ‘Motorizacion’ y para poder usar la herencia debemos añadir después del nombre de la clase la palabra ‘extends’ seguido del nombre de la clase de la que queramos enlazar la herencia, que en este ejemplo es ‘Coche’
  - Dentro de la clase ‘Motorizacion’ hemos creado un método llamado ‘novedad’ que devuelve un mensaje usando los argumentos de la clase ‘Coche’ usando ‘this’
  - Fuera de ambas clases creamos una instancia llamada ‘miMotor’ pasando los dos argumentos a la clase ‘Motorizacion’
  - Por último hacemos dos llamadas usando la instancia ‘miMotor’ a los métodos ‘presentacion’ y ‘novedad’
  - Los mensajes que devuelve son:
    - El DS4 es el nuevo modelo de la marca Citroen
    - El Citroen DS4 usa un motor fabricado por Citroen

Fuentes:

- Platzi
- Free Code Camp
- Developer Mozilla
- Bluuweb GitHub

¿Qué es una promesa JS?

Es un objeto que informa si una operación ha sido positiva o ha dado algún tipo de error. Esta operación puede ser una petición a un servidor, una lectura de un archivo, o cualquier otra operación que necesite un tiempo en ejecutarse por completo.

- Las promesas pueden tener tres estados:
  - Pendiente: Cuando acaba de crearse y aun no se ha ejecutado.
  - Cumplida: Cuando la operación termina de forma positiva, la promesa pasa a estar en estado cumplido.
  - Rechazada: Si la operación falla, la promesa pasa a estar en estado rechazado y se rechaza con un motivo.

Las promesas en JavaScript se crean con 'promise', que toma una función de respuesta con dos parámetros, 'resolve' y 'reject'. La función de respuesta se ejecuta inmediatamente y puede realizar tareas asincrónicas.

Al usarlas se puede encadenar una serie de operaciones de forma más fácil y sencilla de interpretar, se evita el anidamiento excesivo de retornos. Además ofrecen un manejo más sencillo de errores con el método 'catch()'

Cuando estas tareas han terminado se hace una llamada a 'resolve' para avisar de que ha salido bien, o a 'reject' si ha dado algún un error.

- Ejemplo de una promesa que ha sido positiva:

```
const promesaExitosa = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Datos positivos');
  }, 2000);
});

promesaExitosa.then((mensaje) => {
  console.log(mensaje);
}).catch((error) => {
  console.log(error);
});
```

- Análisis del código:
  - Dentro de la función 'promesa' se usa 'setTimeout' para avisar de una operación que tarda dos segundos en completarse.
  - Después de esos 2 segundos se llama a la función 'resolve' con el argumento 'Datos positivos', lo que indica que la promesa se ha completado bien y ha devuelto datos.
  - Se utiliza el método 'then' en la variable 'promesaExitosa' para cuando la promesa se resuelve correctamente. Dentro de la función de 'then' se recibe el mensaje 'Datos positivos' y se imprime en la consola.
  - Se utiliza el método 'catch' en el caso de que la promesa falle y se llama a la función 'reject' pero no se ejecuta ningún código si algo ha dado un fallo, por lo que no se imprime ningún error en la consola.
  - Este código crea una promesa que se resuelve después de 2 segundos y luego devuelve el mensaje 'Datos positivos' y no hace ninguna operación en el caso de que haya dado fallos en la ejecución.

- Ejemplo de una promesa que ha dado fallo:

```
const promesaFallida = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(new Error('Error'));
  }, 2000);
});

promesaFallida.then((mensaje) => {
  console.log(mensaje);
}).catch((error) => {
  console.log(error.message);
});
```

- Análisis del código:
  - Primero creamos la promesa utilizando ‘new Promise’ en donde recibimos los parámetros ‘resolve’ y ‘reject’
  - En este ejemplo estamos usando ‘reject’ para avisar que ha fallado y estamos pasando un nuevo objeto ‘Error’ con el mensaje ‘Error’.
  - Dentro de la función ‘setTimeout’ estamos avisando de que la operación tarda dos segundos antes de rechazar la promesa con el error.
  - Encadenamos ‘.then()’ y ‘.catch()’ a la promesa ‘promesaFallida’ si la promesa se resuelve correctamente, se ejecutará el retorno dentro del ‘.then()’ pero como en este caso la promesa se rechaza y se ejecutará el retorno dentro del ‘.catch()’.
  - Dentro del ‘.catch()’ estamos devolviendo el mensaje de error utilizando ‘error.message’ y se imprimirá ‘Error’

Es importante tener en cuenta el uso de ‘setTimeout’ para avisar al sistema de cuando se debe dar por error la promesa, en caso de no poner bien el tiempo o de no ponerlo, puede dar a errores y fallos sin que la comunicación tenga nada que ver en el fallo en sí.

Existen diferentes métodos para el uso de promesas:

- .then()
  - Ejecuta el retorno cuando la promesa ha sido positiva.
- .catch()
  - Ejecuta el retorno cuando la promesa ha dado algún error.
- .then()
  - Metodo que sirve igual que ‘.then()’ y ‘.catch()’ dentro del paréntesis de ‘.then()’
- .finally()
  - Ejecuta el retorno final, tanto si la promesa ha sido buena o ha dado error.

Fuentes:

- Arsys
- Javascript Info
- Developer Mozilla

¿Qué hacen async y await por nosotros?

Las palabras clave ‘async/await’ sirven para poder hacer una mejor gestión de las promesas, hacer que sea más sencillo y fácil de interpretar. Evita el uso de excesivos anidamientos a la hora de usar los retornos cuando se crean promesas de la forma tradicional.

‘async’ se usa para declarar una función como asíncrona y por ello la función devolverá una promesa. Dentro de una función ‘async’ se pueden usar la palabra ‘await’ que pausa la ejecución de la función asíncrona hasta que la promesa es resuelta.

Siempre utiliza ‘async’ en combinación con ‘await’ ya que el uso de ‘async’ sin ‘await’ en una función solo se estaría devolviendo una promesa sin hacer ninguna operación real.

Al usar ‘async/await’ es importante manejar los posibles errores que puedan ocurrir. Es recomendable usar una estructura ‘try/catch’ para capturar los errores y gestionarlos de forma adecuada.

Aunque ‘async/await’ simplifica el manejo de promesas es importante recordar que sigue siendo necesario trabajar con promesas en algunos casos.

‘async’ se utiliza para declarar que una función es asíncrona, es decir, que la función retornará una promesa que se resolverá con el valor que retorne la función.

‘await’ se utiliza dentro de una función async para esperar a que una promesa se resuelva antes de continuar con la ejecución de la función. Esto evita el uso excesivo de ‘then()’ y ‘catch()’

- Ejemplo

```
1  function retraso(ms) {
2    return new Promise(resolve => setTimeout(resolve, ms));
3  }
4
5  async function saludo() {
6    await retraso(2000);
7    console.log('Saludos');
8  }
9
10 saludo();
```

- Análisis del código
  - Tenemos una función llamada ‘retraso’ que devuelve una promesa que resuelve después de un tiempo que se le pasa como argumento llamado ‘ms’
  - Tenemos una segunda función llamada ‘saludo’ que es asincrónica usando ‘async’ y usa la palabra ‘await’ para avisar de que espera a que la promesa de la función ‘retraso’ se resuelva antes de continuar.
  - Cuando se llama a la función ‘saludo’ esta a su vez llama a la función ‘retraso’ pasándole como argumento ‘2000’ que significa que le dará un retraso de 2 segundos a que la promesa de la función ‘retraso’ se resuelva.
  - La función ‘saludo’ espera esos dos segundos para recibir el resuelto de la función ‘retraso’ y entonces continua a la siguiente línea (En nuestro ejemplo la línea 7) que nos devuelve el mensaje ‘Saludos’
  - Por último tenemos la llamada a la función ‘saludo’ lo que inicia la ejecución de la función y nos tiene que devolver el mensaje ‘Saludos’ tras dos segundos de haber hecho la llamada.

Fuentes:

- Free Code Camp
- Lenguaje JS
- Free Code Camp
- Javascript Info