-Comentar código:

//Comentario
/*Comentario
Multilineal*/

-Declarar variables y asignar valores [var, let y const]:

Acepta los siguientes tipos de datos: undefined, null, boolean, string, symbol, bigint, number, y object.
[var myVar = "Hola";]
[var myNumber = 9;]

También puedes asignar el valor de una variable a otra.
[var secondVar = myVar;]

Las variables que declaramos con [var] tienen un ámbito global (si se declara fuera de una función, lo que implica que esa variable estará disponible para su uso en todo el código) o un ámbito de función (se declara dentro de una función y sólo es accesible desde ella). Además las variables [var] pueden modificarse y volverse a declarar. Cuentan con un mecanismo llamado Hoisting que hace que las variables se muevan a la parte superior del código.

```
[   console.log (saludar);
    var saludar = "dice hola";
```

Se interpretará como:

```
    var saludar;
    console.log(saludar); // saludar is undefined
    saludar = "dice hola";]
```

El problema con [var] es que puedes convertir por error una variable que ya había sido definida en otra parte del codigo.

```
[   var saludar = "hey, hola";
    var tiempos = 4;

    if (tiempos > 3) {
        var saludar = "dice Hola también";
    }

    console.log(saludar) // "dice Hola también"]
```

Las variables que declaramos con [let] tienen un ámbito de bloque {}. Por lo que la variable solo estará disponible dentro de las llaves.

```
[  let saludar = "dice Hola";
   let tiempos = 4;

   if (tiempos > 3) {
       let hola = "dice Hola también";
       console.log(hola);// "dice Hola también"
    }
   console.log(hola) // hola is not defined]
```

Las variables [let] pueden modificarse, pero no pueden volver a declararse.

```
[  let saludar = "dice Hola";
   saludar = "dice Hola también";


   let saludar = "dice Hola";
   let saludar = "dice Hola también"; // error: Identifier 'saludar' has already been declared
]
```

Sin embargo si la variable se define en distintos ámbitos no arrojará error, ya que son tratadas como variables diferentes:

```
[   let saludar = "dice Hola";
    if (true) {
       let saludar = "dice Hola también";
       console.log(saludar); // "dice Hola también"
    }
    console.log(saludar); // "dice Hola"]
```

El Hoisting de [let], al igual que en [var] las declaraciones [let] se elevan a la parte superior. Pero a diferencia de [var] que se inicializa como undefined, [let] no se inicializa. Por lo que su uso sin declararla dará un "Reference Error".

Las variables declaradas con [const] mantienen un valor constante, y tienen similitudes con las [let]. Tienen ámbito de bloque. Las variables [const] no pueden modificarse ni volver a declararse. Por lo que no podremos hacer esto:

```
[   const saludar = "dice Hola";
    saludar = "dice Hola también";// error: Assignment to constant variable. ]
```

Por tanto, las declaraciones [const] deben ser inicializadas en el momento de la declaración. Al crear objetos con [const] el comportamiento es algo diferente. No se puede actualizar el objeto en sí, pero si sus propiedades.

```
[   const saludar = {
       mensaje: "dice Hola",
       tiempos: 4
    }
```

No podemos hacer esto:

```
   saludar = {
      palabras: "Hola",
      número: "cinco"
   } // error:  Assignment to constant variable.
```

Podemos hacer esto:

```
  saludar.mensaje = "dice Hola también";]
```

-Operaciones con números:

```
[   const suma = 3 + 4; //Suma
    const diferencia = 5 - 2; //Resta
    const multi = 2 * 3; //Multiplicación
    const div = 8 / 3; //División
    const reminder = 5 % 2; //Encuentra el resto de la división

    let number = 0;
    number++; //Incrementa en uno la variable
    number--; //Decrementa en uno la variable

    let myVar = 1;
    myVar += 4; //Asignación compuesta con adición aumentada
    myVar -= 3; //Asignación compuesta con resta aumentada
    myVar *= 2; //Asignación compuesta con multiplicación aumentada
    myVar /= 4; //Asignación compuesta con división aumentada ]
```

-Escapar caracteres en strings:

```
[const sampleStr = "Alan said, \"Peter is learning JavaScript\"."; //Usamos \" ]
```

También podemos anidar comillas dobles dentro de comillas simples:

```
[var myStr = '<a href="http://www.example.com" target="_blank">Link</a>';]
```

Otros caracteres que pueden ser escapados en un string:

\'	comilla simple	\"	comilla doble
\\	barra invertida	\n	línea nueva
\t	tabulador	\r	retorno del carro
\b	límite de palabra	\f	fuente de formulario

-Concatenar cadenas:

```
[  var myName = "Bobby";
   var myFavLetter = "B";
   var sentence = "Hello, I'm " + myName + ". My favorite letter is " + myFavLetter + ".";
   //Podemos concatenar cadenas con +. Permite insertar variables
   var myFullName = myName += "Fisher"; //Concatena una cadena al final de una variable]
```

-Encuentra la longitud de una cadena:

```
[console.log("Bobby Fisher".length); //Cuenta los caracteres, tiene en cuenta espacios]
```

Para acceder a un carácter concreto de un string podemos usar la noción de corchetes []. Las cadenas tienen base 0, por lo que el primer carácter será[0]:

```
[  const firstName = "Ada";
   const secondLetterOfFirstName = firstName[0]; //Retorna A
   const lastLetterOfFirstName = firstName[firstName.lenght - 1] //Con .length - 1 podremos acceder al último carácter de la cadena ]
```

-Arreglos [array]:

Los arrays nos permiten almacenar varias variables en un solo lugar, usando notación de corchetes. Además pueden ser anidados uno dentro de otro. Para acceder a los datos dentro de un arreglo también podemos utilizar índices [].

```
[  const sandwich = ["peanut butter", "jelly", "bread"];
   const teams = [["Bulls", 23], ["White Sox", 45]]; //Arreglo anidado
   const var = teams[1][0]; //devolverá White Sox]
```

Los datos dentro de los arreglos son mutables lo que nos permite cambiarlos libremente incluso si fueron declarados con [const].

```
[  const myArray = [10, 11, 12];
   myArray[0] = 9; //Modificamos el valor del primer carácter de la cadena ]
```

-Manipulación de Arrays [.push, .pop, .shift, .unshift];

[const arr1 = [1, 2, 3];
arr1.push(4); //Añade datos al final de una cadena [1, 2, 3, 4]

const arr2 = [1, 2, 3];
const lastValue = arr2.pop(); //Elimina el último valor de un arreglo y devuelve ese elemento, lo que permite asignarlo a una variable, por ejemplo.

const arr3 = [1, 2, 3];
const firstValue = arr3.shift(); //Elimina el primer valor de un array y lo devuelve.

const arr4 = [1, 2, 3];
arr4.unshift(0); // Añade un elemento al principio de una cadena [0, 1, 2, 3]]

-Funciones:

Las funciones son partes reutilizables de código que pueden ser llamadas más adelante. Las funciones admiten parámetros. Estos son variables que actúan como marcadores de posición para los valores que deben ser introducidos en una función cuando se llama.

[function functionName(myName) { //Creas la función con el parámetro myName
console.log("Hello World " + myName); //Este es el código que se ejecuta cuando la llames. Una cadena de texto + una variable
}
functionName(Bobby); //Llamas a la función asignado “Bobby” como valor de la variable]

Puedes utilizar una declaración de devolución [return] para enviar un valor fuera de una función. Una vez se alcanza una sentencia [return], la ejecución de la función se detiene y el control se devuelve a la ubicación de la llamada:

[function plusThree(num) {
return num + 3; //La función suma 3 al argumento dado y lo devuelve
}

const answer = plusThree(5); //Se asigna el valor de la función anterior a una constante]

Es posible tener variables locales y globales con el mismo nombre. Cuando haces esto, la variable local tiene precedencia sobre la variable global. Una función puede incluir la declaración de devolución [return] pero no tiene por qué hacerlo. En el caso de que la función no tenga una declaración de devolución [return], cuando la llames, la función procesa el código interno, pero el valor devuelto es undefined (indefinido).

-Valores booleanos:

Otro tipo de datos es el Booleano. Los booleanos solo pueden ser uno de dos valores: true (verdadero) o false (falso). Básicamente son pequeños interruptores de encendido, donde

true es encendido y false es apagado. Estos dos estados se excluyen mutuamente. Estos no pueden ir entre comillas o serán interpretados como strings.
-Lógica condicional [if]:

Las sentencias [if] son utilizadas para tomar decisiones en el código. El [if] ejecutará el código entre llaves cuando la condición, entre los paréntesis sea verdadera (true).

```
[  function test (myCondition) { //Se crea una función con un argumento
     if (myCondition) {  //Se condiciona al valor de ese argumento
       return "It was true";  //Condición para argumento = true
      }
   }]
```

Hay muchos operadores de comparación en JavaScript que devuelven un valor booleano, por lo que pueden ser utilizados para crear la condición de un [if].

[== Compara dos valores y devuelve true si son equivalentes. No tiene en cuenta el tipo de dato]
[=== Comparación estricta. Tiene en cuenta el tipo del dato]
[> Si el número de la izquierda es mayor que el de la derecha devuelve true]
[>= Si el número de la izquierda es mayor o igual que el de la derecha devuelve true]
[< Si el número de la izquierda es menor que el de la derecha devuelve true]
[<= Si el número de la izquierda es menor o igual que el de la derecha devuelve true]

[&& Sirve para probar más de una cosa. Devuelve true cuando los comparadores a izquierda y derecha son verdaderos]
[|| Devolverá true cuando alguno de los comparadores sea verdadero]

Cuando la condición de una sentencia [if] es falsa, normalmente no pasará nada. La sentencia [else] nos permite ejecutar un bloque de código alternativo.

```
[   if (num > 10) {
     return "Bigger than 10"; //Condición = true
    } else {
    return "10 or Less";  //Condición = false
    }]
```

Cuando tienes múltiples condiciones que necesitan ser resueltas, puedes encadenar sentencias [if] y [else if].

```
[   if (num > 15) {   //Primera condición
      return "Bigger than 15";
   } else if (num < 5) {  //Segunda condición
      return "Smaller than 5";
   } else {   //Anteriores condiciones = false
      return "Between 5 and 15";
   }]
```

-Seleccionando entre múltiples opciones [switch]:

Si tienes muchas opciones para elegir, usa una declaración [switch]. Esta prueba prueba un valor y puede tener muchas sentencias [case] que definen los valores posibles. Cuando encuentra un valor igual al [case] ejecuta el código hasta encontrar un [break]. [switch] nos permite usar una acción predeterminada para los valores no definidos en [case]. Para ello utilizaremos [default].

```
[   switch (fruit) {  //Argumento fruit
      case "apple": //Cuando fruit = apple
        console.log("The fruit is an apple");  //Ejecuta
      break; //Se detiene
      case "orange": //Cuando fruit = orange
         console.log("The fruit is an orange"); //Ejecuta
      break;  //Se detiene
      default:  //Cuando fruit no es igual a ningún case anterior
         console.log(“The fruit is different”); //Ejectua
      break; //Se detiene}]
```

El seleccionador de opciones múltiples [switch] permite asignar la misma salida a varias sentencias [case] distintas.

```
[   let result = "";
    switch (val) {
       case 1:
       case 2:
       case 3:
          result = "1, 2, or 3"; //Se ejecuta cuando [val] sea igual a 1, 2 o 3.
       break;
       case 4:
          result = "4 alone";  //Se ejecuta cuando [val] sea igual a 4.
        break;}]
```

-Construcción de objetos en JavaScript:

Los objetos son similares a los arreglos. Excepto en que en lugar de usar índices [i] para acceder y modificar sus datos, accedes a los datos del objeto a través de sus propiedades. Los objetos son útiles para almacenar datos de forma estructurada:

```
[   const cat = {
      "name": "Whiskers",
      "legs": 4,
      "tails": 1,
      "the enemies": ["Water", "Dogs"]};]
```

Si tu objeto tiene propiedades que no son cadenas, JavaScript las convertirá en cadenas automáticamente.

Para acceder a las propiedades de un objeto, puedes usar tanto la notación de puntos [.], como la de corchetes []. La notación de puntos se utiliza cuando conocemos el nombre de la propiedad a la que queremos acceder. Si la propiedad del objeto a la que deseas acceder tiene un espacio en el nombre, necesitarás usar la notación de corchetes [].

```
[   const catName = cat.name; //Devuelve la propiedad name en el objeto cat
    const catEnemies = cat[“the enemies”]; //Devuelve la propiedad “the enemies” en el objeto cat]
```

Otro uso de la notación de corchetes es acceder a una propiedad que está almacenada como el valor de una variable.

```
[   const myCatInfo = “legs”;  //Asigna el nombre de la propiedad a una variable
    const catInfo = cat[myCatInfo];  //Accede a la información del objeto usando la variable]
```

Los objetos creados en JavaScript pueden ser actualizados en cualquier momento. Para ello se puede usar tanto la notación de puntos, como la de corchetes.

```
[   const ourDog = {
      "name": "Camper",
      "legs": 4,
      "tails": 1,
      "friends": ["everything!"]
    };
    ourDog.name = “Happy Camper”;  //Actualización con notación de puntos
    ourDog[“name”] = “More Happy Camper”;  //Actualización con notación de corchetes]
```

También puedes añadir nuevas propiedades a un objeto existente de la misma manera que lo modificarías. O eliminarlas usando [delete].

```
[   ourDog.bark = “woof”;  //Crea una nueva propiedad
    delete ourDog.tails;  //Elimina una propiedad existente]
```

Los objetos pueden ser considerados almacenes de clave/valor. Por lo que puedes utilizar un objeto para hacer búsquedas de valores que antes realizabas con declaraciones [switch] o cadenas de [if/else]. Es de mucha utilidad cuando los datos están limitados a cierto rango.

Para verificar las propiedades de un objeto podemos utilizar el método [.hasOwnProperty(propertyName)]. Este devuelve un valor booleano (true/false).

```
[   ourDog.hasOwnProperty(“name”); //Devuelve true
    ourDog.hasOwnProperty(“age”); //Devuelve false]
```

Estos dos códigos son equivalentes:

```
function phoneticLookup(val) {
  let result = "";

  switch(val) {
    case "alpha":
      result = "Adams";
      break;
    case "bravo":
      result = "Boston";
      break;
    case "charlie":
      result = "Chicago";
      break;
    case "delta":
      result = "Denver";
      break;
    case "echo":
      result = "Easy";
      break;
    case "foxtrot":
      result = "Frank";
  }

  return result;
}
```

```
function phoneticLookup(val) {

var result = "";

  var lookup = {
    "alpha": "Adams",
    "bravo": "Boston",
    "charlie": "Chicago",
    "delta": "Denver",
    "echo": "Easy",
    "foxtrot": "Frank"
  };

  result = lookup[val];
}
```

Los objetos en JavaScript son una forma de manejar datos de manera flexible. Permiten combinaciones arbitrarias de: strings, números, arreglos, funciones e incluso otros objetos. Se puede acceder a las subpropiedades de un objeto encadenando notación de puntos y corchetes. Podemos acceder a las propiedades de arreglos anidados encadenando [].

Objetos Anidados

```
 const ourStorage = {
      "desk": {
         "drawer": "stapler"
        },
      "cabinet": {
         "top drawer": {
              "folder1": "a file",
              "folder2": "secrets"
           },
          "bottom drawer": "soda"
         }  };
```

Arreglos Anidados

```
const myPlants = [
  {
    type: "flowers",
    list: ["rose", "tulip", "dandelion"]
  },
  {
    type: "trees",
    list: ["fir", "pine", "birch"]
  }
];
```

ourStorage.cabinet["top drawer"].folder2; ourStorage.desk.drawer;	const secondTree = myPlants[1].list[1];

-Bucle [while]:

Ejecuta una condición específica mientras esta sea verdadera.

```
[   const ourArray = []; //Se crea un array
    let i = 0;  //Se crea una variable y se inicializa a 0

    while (i < 5) {  //Se ejecuta el código entre {}, mientras [i] sea menor a 5.
        ourArray.push(i); //Inserta el valor de [i] en el array
         i++; //Suma 1 a [i]  }]
```

-Bucle [for]:

El bucle [for] se ejecuta un por un número de veces específico. Los bucles [for] se declaran con tres expresiones, separadas por punto y coma (;). Tal qué [for (a; b; c)], donde [a] es la sentencia de iniciación (que solo se ejecutará una vez, antes del comienzo del bucle), [b] el condicional (que es evaluado al principio de cada iteración y continuará mientras sea “true”) y [c] la expresión final (se ejecuta al final de cada iteración, antes de la siguiente comprobación del condicional, suele utilizarse para incrementar o disminuir el contador).

```
[    const ourArray = [];  //Se crea el array

     for (let i = 0; i < 5; i++) { //Se inicializa i = 0, mientras i sea menor a 5, incrementa i en 1.
       ourArray.push(i); //Añade el valor de i al array }

    for (let i = 0; i < 10; i += 2) { // En este caso el incremento será de 2, arroja números pares
       ourArray.push(i);
    }]
```

El bucle [for] nos permite iterar a través del contenido de un arreglo.

```
[   const arr = [10, 9, 8, 7, 6];

    for (let i = 0; i < arr.length; i++) { //La condición es que i sea menor a la longitud del arreglo
       console.log(arr[i]); //Accede a todos los valores utilizando la notación de corchetes []
    }]
```

Para recorrer un arreglo multidimensional, se puede utilizar la misma lógica y anidar bucles [for].

```
[   const arr = [    //Creamos un array multidimensional
      [1, 2], [3, 4], [5, 6]
    ];

    for (let i = 0; i < arr.length; i++) {  //Accedemos al primer índice
      for (let j = 0; j < arr[i].length; j++) {  //Accedemos al segundo índice
        console.log(arr[i][j]);
      }
    }]
```

-Bucle [do…while]:

Este bucle hace una pasada al código sin importar qué (do), y luego continúa ejecutandolo en bucle mientras (while) la condición sea verdadera.

```
[    const ourArray = [];  //Crea un array
    let i = 0;   //Crea una variable y la inicializa a 0

    do {
      ourArray.push(i);  //Inserta el valor de i en el array
       i++;   //Suma 1 a i
    } while (i < 5);  //Repite el código anterior mientras el condicional sea verdadero]
```

-Funciones recursivas:

Una función recursiva es una función que se llama a sí misma cuando se ejecuta. Debe estar compuesta por dos partes; un caso base (es el que nos dice cuando podemos acabar con la recursividad) y un caso recursivo (es donde la función se llama a sí misma). Pueden tener más de un caso base y más de un caso recursivo.

Buscamos crear una función recursiva que devuelva la suma de los [n] primeros elementos de un arreglo. Los parámetros serán el arreglo y una variable [n].

```
[     function sum(arr, n) { //Se crea una función con dos parámetros
        if(n <= 0) {     //Si n es menor o igual a 0
          return 0;   //Devuelve 0
        } else {    //Si no
          return sum(arr, n - 1) + arr[n - 1];  }//Devuelve el resultado de volver a llamar a la
```

función con n - 1 y sumarle el valor de arr[n - 1]. Volverá a ejecutarse mientras no se cumpla la condición n menor o igual a 0.

-Generar números aleatorios en JavaScript [Math.random()]:

Los números aleatorios son útiles para crear comportamientos aleatorios. [Math.random()] devuelve un número decimal aleatorio entre 0 (inclusive) y 1 (excluido), por lo que podrá devolver 0 pero no 1. Las funciones se ejecutan antes de que se ejecute el [return].

Para generar un número entero aleatorio, solo tendremos que multiplicar el resultado por un número entero y redondear el resultado con la función [Math.floor()]. Esta función redondea hacía abajo, al número entero más cercano.

[Math.floor(Math.random() * 11); //Entrega un número entero, aleatorio entre 0 y 10]

Puede que en lugar de querer generar un número aleatorio entre 0 y el parámetro dado, queramos hacerlo en un rango concreto. Para ello definiremos el número min y max.

[Math.floor(Math.random() * (max - min + 1)) + min; //Entrega un entero aleatorio entre dos números.]

-Convertir strings a números:

La función [parseInt()] analiza una cadena y devuelve un número entero. Si el primer carácter de la cadena no puede ser convertido, la función retornará NaN.

[const a = parseInt(“009”); //Convierte la cadena “009” a 9]

La función [parseInt()] admite otro parámetro, a parte del string que vamos a convertir. El parámetro [radix] se utiliza para especificar la base numérica de la cadena que se va a convertir (por defecto esta será 10). Por ejemplo podríamos interpretar una cadena como un binario (base 2).

[parse int(“101”, 2); //Devolverá 5, ya que 101 es 5 en binario]

-Uso del operador condicional (ternario):

El operador ternario puede utilizarse como una expresión [if/else] de una sola línea. La sintaxis es [a ? b : c], donde [a] es la condición, [b] el código a ejecutar cuando la condición es verdadera, y [c] es el código que se ejecutará cuando la condición es falsa. Los siguientes códigos son equivalentes

[function findGreater(a, b) {
if(a > b) {
return "a is greater";
} else {
return "b is greater or equal";
}
}]
	function findGreater(a, b) {
	return a > b
	? "a is greater"
	: "b is greater or equal";
	}

Los operadores condicionales ternarios pueden ser encadenados para comprobar múltiples condiciones como haría un bucle [if/else if].

function findGreaterOrEqual(a, b) {
  if (a === b) {
  return "a and b are equal";
  }
  else if (a > b) {
  return "a is greater";
  }
  else {
  return "b is greater";
  }
}
	function findGreaterOrEqual(a, b) {
	  return (a === b) ? "a and b are equal"
	  : (a > b) ? "a is greater"
	  : "b is greater";
	}