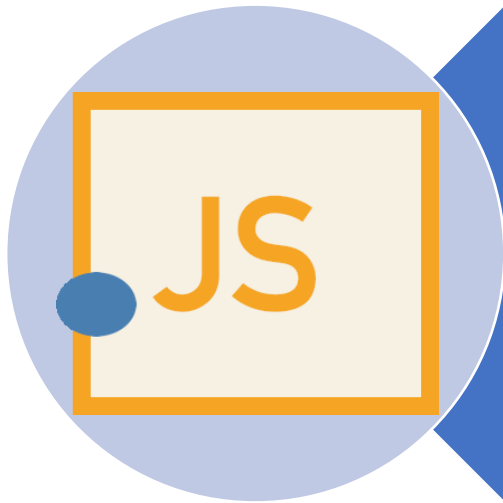


# Desarrollo Web en Entorno Cliente

## Tema 7





Valor y Referencia.  
Más Operadores.  
Estructuras de  
control y bucles.

# [paso por valor y referencia]

“Paso por valor” significa que se crea una copia del valor de la variable cuando se asigna o se pasa como argumento de una función.

El valor de la variable original y la que se “pasa” son independientes, por lo que los cambios que se hagan en una no afectan a la otra.

```
let a=10;  
let b=a;  
  
a=11;  
  
console.log({a});  
console.log({b});
```

```
▶ {a: 11}  
▶ {b: 10}
```

“Paso por referencia” significa que no se copia el valor de la variable, lo que se copia es la dirección de memoria donde se encuentra la variable cuando se asigna o se pasa como argumento de una función.

El valor de la variable original y la que se “pasa”, son la misma, son dependientes, por lo que los cambios se realicen en una sí afectan a la otra.

```
let a=[10];  
let b=a;  
  
a[0]=11;  
  
console.log({a});  
console.log({b});
```

```
▼ {a: Array(1)} ⓘ  
  ▶ a: [11]  
  ▶ [[Prototype]]: Object  
▼ {b: Array(1)} ⓘ  
  ▶ b: [11]  
  ▶ [[Prototype]]: Object
```

# [paso por valor y referencia]

En el siguiente ejemplo, podría pensarse que, al cambiar dentro de la función el valor de la variable *miVariable* este cambio es permanente (fuera de la función).

Pero realmente la variable *miVariable* se pasa a la función “por valor”, no “por referencia”, esto significa que se crea una copia de la variable, incluido su valor, por lo que no se trabaja con *miVariable*, sino que se hace con una copia temporal de la misma que al salir de la función se destruye.

```
let miVariable=0;

function miFunción(arg1){
    arg1+=1;
    console.log(`Dentro de la función ${arg1}`);
}

miFunción(miVariable);
console.log(`Fuera de la función ${miVariable}`);
```

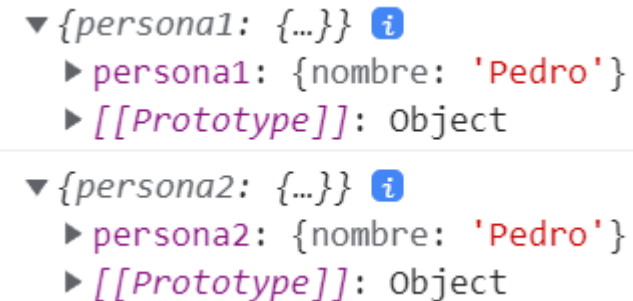
|                        |
|------------------------|
| Dentro de la función 1 |
| Fuera de la función 0  |

# [paso por valor y referencia]

JS pasa por valor los tipos de datos primitivos, pero cuando trabaja con tipos de datos no primitivos (*Object* y *Array*, en principio, pero hay que recordar que en JS todo son objetos menos los tipos de datos primitivos) los pasa por referencia.

Esto implica que no se copia el valor en sí, si no una referencia a través de la cual se accede al valor original.

```
const persona1={nombre:'Jorge'};  
const persona2=persona1;  
  
persona2.nombre='Pedro';  
  
console.log({persona1});  
console.log({persona2});
```



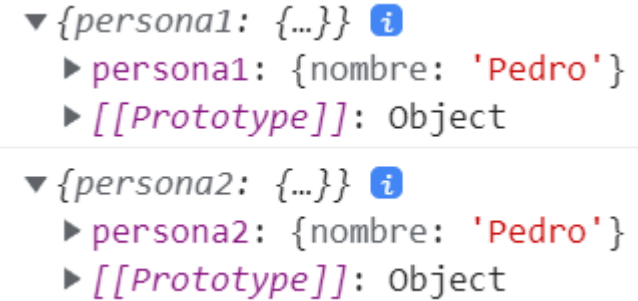
```
▼ {persona1: {...}} ⓘ  
  ► persona1: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object  
  
▼ {persona2: {...}} ⓘ  
  ► persona2: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object
```

# [paso por valor y referencia]

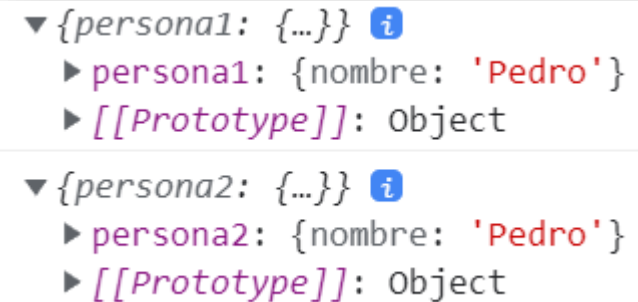
Como JS pasa los tipos de datos no primitivos por referencia, esto puede dar “problemas”:

```
const persona1={nombre:'Jorge'};  
  
const persona2=Object.assign(persona1);  
  
persona2.nombre='Pedro';  
console.log({persona1});  
console.log({persona2});
```

```
const persona1={nombre:'Jorge'};  
  
const persona2= new Object(persona1);  
persona2.nombre='Pedro';  
  
console.log({persona1});  
console.log({persona2});
```



```
▼ {persona1: {...}} ⓘ  
  ► persona1: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object  
  
▼ {persona2: {...}} ⓘ  
  ► persona2: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object
```

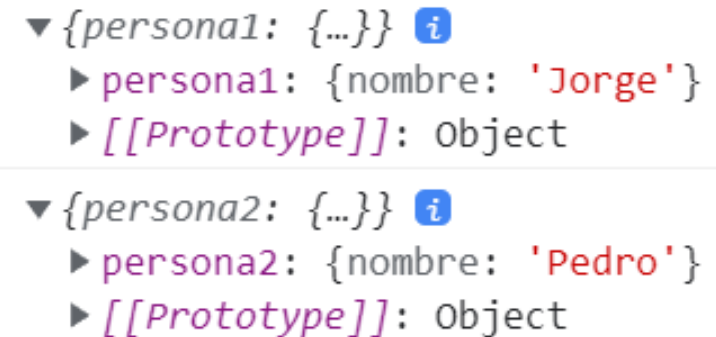


```
▼ {persona1: {...}} ⓘ  
  ► persona1: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object  
  
▼ {persona2: {...}} ⓘ  
  ► persona2: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object
```

# [paso por valor y referencia]

Para solucionarlo se puede usar el *Spread Operator*.

```
const persona1={nombre:'Jorge'};  
const persona2={...persona1};  
  
persona2.nombre='Pedro';  
  
console.log({persona1});  
console.log({persona2});
```



```
▼ {persona1: {...}} ⓘ  
  ► persona1: {nombre: 'Jorge'}  
  ► [[Prototype]]: Object  
  
▼ {persona2: {...}} ⓘ  
  ► persona2: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object
```

# [rest operator]

*Rest Operator* es muy parecido en sintaxis al *Spread Operator* pero tiene la funcionalidad opuesta al *Spread Operator*.

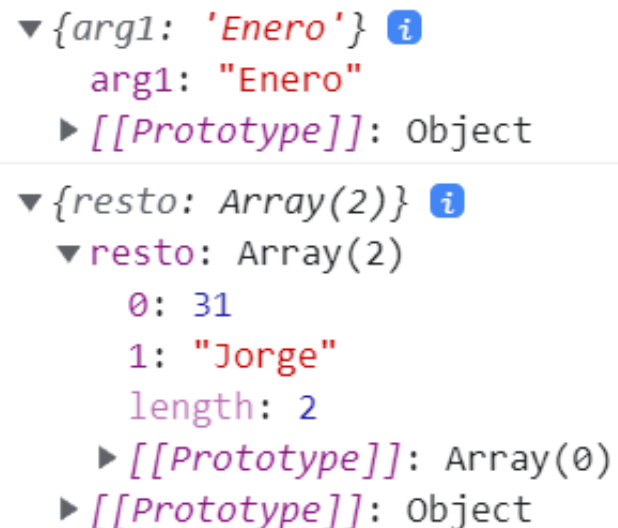
Si *Spread Operator* desenrollaba, esparcía, propiedades y valores, *Rest Operator* los agrupa, los recoge.

La sintaxis es la misma en uno y en otro, se utilizan los tres puntos `...`, lo que cambia es el contexto donde se utiliza.

```
const mes='Enero', días=31, nombre='Jorge';

const miFunción=(arg1, ...resto)=>{
  console.log({arg1});
  console.log({resto});
};

miFunción(mes,días,nombre);
```



```
▼ {arg1: 'Enero'} ⓘ
  arg1: "Enero"
  ► [[Prototype]]: Object

▼ {resto: Array(2)} ⓘ
  ▼ resto: Array(2)
    0: 31
    1: "Jorge"
    length: 2
    ► [[Prototype]]: Array(0)
  ► [[Prototype]]: Object
```



# [rest operator]

*Rest Operator* se utiliza sobre todo con funciones, en este caso, se le conoce también como *Rest parameter*.

```
function suma(...args){  
  return args.reduce((anterior,actual)=>anterior+actual);}
  
console.log(suma(1,2,3));  
console.log(suma(1,2,3,4));  
console.log(suma(1020,52,553,2,97));
```

|      |
|------|
| 6    |
| 10   |
| 1724 |

Aunque también se utiliza en otras partes, como en la desestructuración de un array (en aquel momento se llamó de forma “*incorrecta*” *Spread Operator*):

```
const meses=["Enero","Febrero","Marzo","Abril",  
  ,"Mayo","Junio","Julio","Agosto"];
  
const [, ,mes3,mes4, ...restoMeses]=meses;
```

## [rest operator]

Cuando es el *Rest Operator*, el operador que agrupa elementos debe ponerse en último lugar, pues va a recoger el resto de elementos que quedan, en otro lugar dará error:

```
const meses=["Enero","Febrero",  
"Marzo","Abril","Mayo","Junio",  
"Julio","Agosto"];  
  
const [, ,mes3,mes4, ...restoMeses, últimoMes]=meses;
```

✖ Uncaught SyntaxError: Rest element must be last element

# [rest operator]

```
const persona1={nombre: 'Jorge'};  
const persona2=persona1;
```

```
function cambiarNombre(persona,nombre)  
{persona.nombre=nombre;}
```

```
cambiarNombre(persona2, 'Pedro');  
console.log({persona1});  
console.log({persona2});
```

```
▼ {persona1: {...}} ⓘ  
  ► persona1: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object  
  
▼ {persona2: {...}} ⓘ  
  ► persona2: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object
```

```
let persona1={nombre: 'Jorge'}, persona2=persona1;  
console.log('Antes\n',{persona1}, '\n',{persona2});
```

```
function cambiarNombre(persona,nombre) {  
  persona.nombre=nombre;  
  return persona;}  
  
persona2=cambiarNombre({...persona2}, 'Pedro');
```

```
console.log('Después\n',{persona1}, '\n',{persona2});
```

```
Antes  
▼ {persona1: {...}} ⓘ  
  ► persona1: {nombre: 'Jorge'}  
  ► [[Prototype]]: Object  
▼ {persona2: {...}} ⓘ  
  ► persona2: {nombre: 'Jorge'}  
  ► [[Prototype]]: Object  
  
Después  
▼ {persona1: {...}} ⓘ  
  ► persona1: {nombre: 'Jorge'}  
  ► [[Prototype]]: Object  
▼ {persona2: {...}} ⓘ  
  ► persona2: {nombre: 'Pedro'}  
  ► [[Prototype]]: Object
```

# [rest operator]

Por último, como la sintaxis es la misma, los tres puntos . . . , puede resultar confuso saber cuál es cuál.

Diremos que es el *Rest Operator* o *Rest Parameter* cuando el operador agrupe elementos y diremos que es el *Spread Operator* cuando desenrolle elementos.

¿?

```
let persona1={nombre: 'Jorge'}, persona2=persona1;
console.log('Antes\n',{persona1}, '\n',{persona2});

function cambiarNombre(persona,nombre) {
  persona.nombre=nombre;
  return persona;}

persona2=cambiarNombre({...persona2}, 'Pedro');

console.log('Después\n',{persona1}, '\n',{persona2});
```

## (if...else)

Las sentencias de control dirigen el flujo del programa, el script.

En JS, hasta ahora, las sentencias se ejecutan linealmente, de arriba abajo, exceptuando algunas cosas que ya se han visto que incluyen: el *hoisting* (que se anticipa a la declaración), el bucle *for* (que repite sentencias mientras se cumpla una condición) y las funciones (que ejecutan instrucciones en la parte del código donde son llamadas).

La instrucción *if* ejecuta una sentencia si una condición, colocada a continuación, entre paréntesis, es evaluada como verdadera.

Si se añade la palabra *else*, la sentencia que siga después se puede ejecutar si la condición es evaluada como falsa.

Se pueden ir añadiendo más comprobaciones adicionales para evaluar más condiciones, unas van a estar contenidas dentro de otras, esto se conoce anidamiento.

# [if...else]

Sintaxis:

```
if (condición)
    sentencia1
[else
    sentencia2]
```

```
if (nombre=="Jorge") {
    console.log("Hola Jorge");
    console.log("¿Cómo estas hoy?");
}
else if (nombre=="Juan")
    console.log("Buenas Juan!!");
else
    console.log("Hola " + nombre);
```

*condición* es la expresión a evaluar.

*sentencia1* se ejecuta si *condición* es evaluada como verdadera. Puede ser cualquier sentencia, incluyendo otras sentencias *if* anidadas. Para ejecutar múltiples sentencias, hay que usar un bloque de sentencias, un conjunto de instrucciones colocadas entre llaves {}.

*sentencia2* se ejecuta si *condición* se evalúa como falsa, y se ha colocado la palabra reservada *else*. Puede ser cualquier sentencia, incluyendo un bloque de sentencias, un conjunto de instrucciones colocadas entre llaves {} y otras sentencias *if* anidadas.

# [operador ternario]

El operador ternario es parecido a una sentencia *if...else*

Se evalúa una condición, si se cumple se ejecuta una sentencia y si no se cumple se ejecuta otra sentencia.

La sintaxis es la siguiente:

*(condición)? sentencia1 : sentencia2;*

*condición* es la expresión a evaluar; justo después hay que colocar el símbolo de interrogación ?

*sentencia1* se coloca justo después de la interrogación ? y se ejecuta si condición es evaluada como verdadera.

*sentencia2* se coloca justo después de la sentencia a ejecutar como verdadera y separada de esta por el símbolo dos puntos : y se ejecuta si condición se evalúa como falsa.

# [operador ternario]

```
(nombre=="Jorge")? console.Log("Hola Jorge"): console.Log("Hola " + nombre);
```

Si se quiere usar el operador ternario como un *if* sin *else*, esto es, sin la instrucción a ejecutar en caso de que la condición a evaluar sea false, se puede poner un literal vacío:

```
(nombre=="Jorge")? console.Log("Hola Jorge"): '';
```

En el operador ternario también se pueden anidar condiciones, estas tienen que llevar al final el símbolo de interrogación ? y van separadas por el símbolo de dos puntos :

```
(nombre=="Jorge")? (console.Log("Hola Jorge"),  
console.Log("¿Cómo estas hoy?"))  
:(nombre=="Juan")? console.Log("Buenas Juan!!")  
:console.Log("Hola " + nombre);
```

Hola Jorge

¿Cómo estas hoy?



# [operador ternario]

El operador ternario también se puede utilizar para asignar un valor u otros a diferentes variables y constantes según se compruebe una condición.

```
const nota=5;  
const evaluación=(nota<5)? "Suspenso" : "Aprobado";  
  
console.log(evaluación);
```

Aprobado

De forma parecida, también se puede utilizar funciones anónimas auto invocadas para este mismo.

```
const evaluación=[0, "Aprobado", "Suspenso", (()=>"Aprobado")()];  
  
console.table(evaluación);
```

| (índice) | Valor      |
|----------|------------|
| 0        | 0          |
| 1        | 'Aprobado' |
| 2        | 'Suspenso' |
| 3        | 'Aprobado' |

# [method chaining]

JS dispone de un operador que permite realizar varias llamadas encadenadas a métodos, mediante el símbolo punto ., lo que facilita la simpleza y legibilidad del código. Este mecanismo se conoce como encadenamiento de métodos o *Method Chaining*.

Por ejemplo, los siguientes trozos de código son equivalentes:

```
Objeto.metodoTres(Objeto.metodoDos(Objeto.metodoUno())));
```

```
Objeto.metodoUno().metodoDos().metodoTres();
```

No obstante, debe cumplir que los métodos o funciones que se invocan deben pertenecer al objeto devuelto en la invocación del método llamado previamente.

# [method chaining]

Por ejemplo, los siguientes trozos de código son equivalentes:

```
const cadena = "Hola Mundo";  
const array = cadena.split(" ");  
console.log(array.join(""));
```

```
const cadena = "Hola Mundo";  
console.log(cadena.split(" ").join(""));
```

HolaMundo

Obsérvese, en el ejemplo superior, que el método *split()* devuelve un array de solo lectura (constante) y que el método *join()* es un método de *Arrays*, no de *Strings*.

## [method chaining]

En el ejemplo de la derecha, si *Operaciones* no se devuelve así mismo en cada uno de sus métodos no se podrían encadenar, debido a que se invoca un método perteneciente al valor devuelto por cada método previo.

10

```
const Operaciones={
  resultado: 0,
  sumar: (a=0,b= Operaciones.resultado)=>{
    Operaciones.resultado=a+b;
    return Operaciones;
  },
  restar: (a=0,b= Operaciones.resultado)=>{
    Operaciones.resultado=a-b;
    return Operaciones;
  },
  multiplicar: (a=0,b= Operaciones .resultado)=>{
    Operaciones.resultado=a*b;
    return Operaciones;
  },
  mostrar: ()=>console.log(Operaciones.resultado),
}

Operaciones.sumar(2,3).multiplicar(2).mostrar();
```

# [optional chaining]

Con la especificación ECMAScript 2020 se ha agregado un nuevo operador a JS, el encadenamiento opcional (*optional chaining*), presente en otros lenguajes como Swift, consiste en una interrogación y un punto ?.

Este operador hace más sencillo trabajar con claves o propiedades y métodos de objetos sin tener que preocuparse si han sido definidos.

El operador de encadenamiento opcional ?. permite leer el valor de una propiedad o método de un objeto dentro de un encadenamiento sin tener que validar expresamente cada referencia del encadenamiento.



## Swift

Swift es un lenguaje de programación multiparadigma creado por Apple enfocado en el desarrollo de aplicaciones para iOS y macOS

# [optional chaining]

El operador `?.` funciona de manera similar al operador de encadenamiento `.`, excepto que en lugar de causar un error si una referencia es *nullish* (`null` o `undefined`), la expresión devuelve *undefined*.

Cuando se usa con llamadas a funciones, devuelve *undefined* si la función dada no existe.

```
const persona = {  
  nombre: 'Jorge',  
  email: 'mail@mail.com',  
  // trabajoActual:  
  //   {puesto: 'diseñador'}  
}
```

```
const puesto=persona.trabajoActual.puesto;
```

✖ ▶ Uncaught TypeError: Cannot read properties of undefined (reading 'puesto')

```
const puesto=persona.trabajoActual?.puesto;
```

undefined

# [optional chaining]

El siguiente ejemplo hace uso del operador encadenamiento opcional sobre métodos.

```
const persona = {  
  nombre: 'Jorge',  
  email: 'mail@mail.com',  
  // mostrar: ()=>(`${persona.nombre} - ${persona.email}`),  
}
```

```
console.log(persona.mostrar());
```

✖ ▶ Uncaught TypeError: persona.mostrar is not a function

```
console.log(persona.mostrar?.());
```

undefined

# [optional chaining]

En resumen, la sintaxis del operador de encadenamiento opcional tiene tres formas:

- ❑ `obj?.prop` devuelve el valor de la propiedad `prop` del objeto `obj` si existe, en caso contrario devuelve *undefined*.
- ❑ `obj?.[prop]` devuelve el valor de la propiedad `prop` del objeto `obj` si existe, en caso contrario devuelve *undefined*.
- ❑ `obj.met?.()` llama al método `met()` si dicho método está definido en el objeto `obj`, en caso contrario devuelve *undefined*.



# (operador lógico de asignación nullish)

Se llama valor *nullish* en JS a un valor que es o *null* o *undefined*.

Además, el valor *nullish* siempre es considerado como es falso.

```
const nullish=null;

(nullish)? console.log("verdadero")
: console.log("false");
```

El operador lógico de asignación *nullish* `??=` asigna el valor únicamente si la variable es tiene un valor *nullish* (*null* o *undefined*).

```
let valor;

valor ??= 1;
console.log(valor);

valor ??= 2;
console.log(valor);
```

|   |
|---|
| 1 |
| 1 |

# [operador coalescente nullish]

El operador de coalescencia (fusión o aglomeración) es un operador que se corresponde con dos símbolos de interrogación `??`.

Es un operador lógico que devuelve su operando del lado derecho cuando su operando del lado izquierdo es *nullish* (*null*, *false* o *undefined*) y, en caso contrario, devuelve su operando del lado izquierdo. Este operador no se puede combinar directamente con los operadores lógicos: `&&` y `||`.

En el siguiente ejemplo a la constante *nombre* se le asignará el valor que haya tecleado el usuario en la ventana de *prompt*, pero si el usuario pulso el botón *Cancelar* se le asignará *"Invitado"*.

```
const respuesta = prompt("Dime tu nombre");  
const nombre = respuesta ?? "Invitado";  
  
console.log(`Hola ${nombre}!`);
```

# [switch]

La sentencia *switch* evalúa una expresión y compara el valor de esa expresión con una serie de valores, si alguno coincide ejecuta las instrucciones asociadas a ese caso.

Se debe poner la palabra reservada *break* al final de cada *case* para evitar que a partir de ese punto no siga ejecutando el resto de instrucciones contenidas en los diferentes *case*

*switch* resulta más útil que un gran número de *if...else* anidados para aquellas situaciones en las que se trata de comparar si una expresión es igual a un valor determinado.

La sintaxis es la siguiente:

```
switch (expresión)
{
    case valor1:
        //Instrucciones a ejecutar si coincide con el valor1
        [break;]

    case valor2:
        //Instrucciones a ejecutar si coincide con el valor2
        [break;]
    ...

    default:
        //Instrucciones a ejecutar cuando no coincide con
        //ningún valor
        [break;]
}
```

# [switch]

En la sentencia *switch* se compara la *expresión*, si es igual al *valor1* se ejecutan las instrucciones contenidas a continuación hasta que encuentre un *break*.

Si no coincide con *valor1* se evalúa el siguiente *case*, si coincide con *valor2* se ejecutan las instrucciones contenidas a continuación hasta que encuentre un *break*.

De nuevo, si no coincide con *valor2* se evalúa el siguiente *case*.

Hasta que llegue a *default*, por defecto, que se ejecutará si ningún *case* ha sido igual a la expresión a evaluar, colocada justo detrás de la palabra *switch* entre paréntesis.

*default* puede no aparecer en la sentencia *switch* porque es opcional.

```
switch (expresión)
{
    case valor1:
        //Instrucciones a ejecutar
        [break;]

    case valor2:
        //Instrucciones a ejecutar    [break;]
        ...

    default:
        //Instrucciones a ejecutar //ningún valor
        [break;]
}
```

# [switch]

Estos dos trozos de código hacen lo mismo, en el ejemplo de la izquierda se utiliza un *if...else* y en el de la derecha se utiliza un *switch*, pero el resultado es el mismo.

```
if (nombre=="Jorge")
{
    console.Log("Hola Jorge");
    console.Log("¿Cómo estas hoy?")
}
else if (nombre=="Juan")
    console.Log("Buenas Juan!!");
else
    console.Log("Hola " + nombre);
```

```
switch (nombre)
{
    case "Jorge":
        console.Log("Hola Jorge");
        console.Log("¿Cómo estas hoy?");
        break;

    case "Juan":
        console.Log("Buenas Juan!!");
        break;

    default:
        console.Log("Hola " + nombre);
}
```

# objetos y arrays como alternativa a *if...else* y *switch*

Se puede utilizar, como alternativa a *if...else* y a *switch*, objetos y arrays para evaluar una expresión y devolver un valor.

Ejemplo:

```
if (mes==1)
    nombreMes="Enero";
else if (mes==2)
    nombreMes="Febrero";
...
else if (mes==12)
    nombreMes="Diciembre";
```

```
switch (mes)
{
    case 1:
        nombreMes="Enero";
        break;
    case 2:
        nombreMes="Febrero";
        break;
    ...
    case 12:
        nombreMes="Diciembre";
        break;
}
```

# objetos y arrays como alternativa a if...else y switch

Utilizando objetos y arrays quedaría así:

```
const nombreMeses={  
  1:"Enero", 2:"Febrero",  
  3:"Marzo", 4:"Abril",  
  5:"Mayo", 6:"Junio",  
  7:"Julio", 8:"Agosto",  
  9:"Septiembre", 10:"Octubre",  
  11:"Noviembre", 12:"Diciembre"};
```

Octubre

```
mes=10;  
console.log(nombreMeses[mes] || "no es un mes válido");
```

```
const nombreMeses=["", "Enero", "Febrero", "Marzo",  
  "Abril", "Mayo", "Junio", "Julio", "Agosto", "Septiembre",  
  "Octubre", "Noviembre", "Diciembre"];
```

```
mes=10;  
console.log(nombreMeses[mes] || "no es un mes válido");
```

Octubre

## objetos y arrays como alternativa a if...else y switch

Consideremos el ejemplo de una tienda que abre a las 9-14 horas los días de diario, pero los sábados abre a las 10-13 horas y los domingos no abre :

```
if (día>=1 && día<=5)
{
    if (hora>=9 && hora<=14)
        console.log("Estamos abiertos");
    else
        console.log("Estamos cerrados");
}
else if (día==6)
{
    if (hora>=10 && hora<=13)
        console.log("Estamos abiertos");
    else
        console.log("Estamos cerrados");
}
else
    console.log("Estamos cerrados");
```



## objetos y arrays como alternativa a if...else y switch

Se puede utilizar arrays para comprobar el día y la hora del ejemplo con el método *include* de los *Array Methods*:

```
if ([1,2,3,4,5].includes(día) && [9,10,11,12,13,14].includes(hora))  
    console.log("Estamos abiertos");  
else if (día==6 && ([10,11,12,13].includes(hora)))  
    console.log("Estamos abiertos");  
else  
    console.log("Estamos cerrados");
```

# [bucles e iteradores]

Los bucles son estructuras presentes en los lenguajes de programación que permiten ejecutar una serie de instrucciones mientras se cumpla una determinada condición.

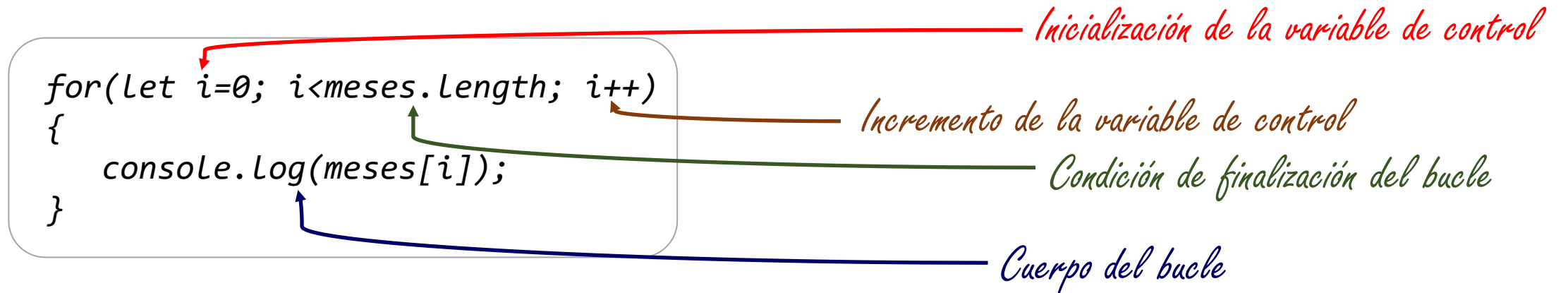
En el tema de los arrays ya se presentó una de estas estructuras, el bucle *for*, que resultaba muy útil para mostrar y actuar sobre cada uno de los elementos de un array.

A veces, se utiliza de forma análoga el término bucle o iterador, no obstante, un iterador hace referencia a una construcción que se utiliza para recorrer un tipo de colección o una lista, como es, por ejemplo, en JS, un array.

Realmente, también ya se ha visto algún iterador, en concreto *forEach* cuando se estudió los arrays. En aquel momento se definió como un método para iterar o recorrer los arrays, que es justamente lo que se ha definido como iterador.

# [for]

Repasando brevemente la estructura del bucle *for*, también llamado *for Loop*:



Mientras se cumpla la condición `i<meses.length` se ejecuta, en cada iteración, las instrucciones contenidas en el cuerpo del bucle.

## [for]

```
const esPrimo=numero=>{  
  if (numero<=2) return true;  
  for(let i=2; i<numero;i++)  
    if (numero%i==0) return false;  
  return true;  
}  
const número=71;  
console.log(`El número ${número} ${esPrimo(número)? "sí": "no"} es primo`);
```

El número 71 sí es primo

El código superior es un ejemplo del uso de bucle *for* para saber si un número es primo o no.

En ciertas ocasiones, conviene saltarse una iteración y pasar a la siguiente, para ello existe la palabra reservada *continue*.

Cuando se ejecuta la instrucción *continue* en un bucle automáticamente finaliza la iteración en curso, no el bucle, a continuación, se incrementa la variable de control, se comprueba la condición del mismo y si se cumple comienza una nueva iteración.

## [for]

La sentencia *continue* puede incluir opcionalmente una etiqueta que permite saltar a la siguiente iteración del bucle etiquetado en vez del bucle actual.

En este caso, la etiqueta tiene que estar un nivel superior y fuera del anidamiento donde está la sentencia *continue*.

```
De los Números: 100,22,66,54,21  
son divisibles por 22: 22,66
```

```
function mostrarDivisibles (números, divisible)  
{  
    let divisibles=[];  
  
    for (let i=0;i<números.length; i++)  
    {  
        if (números[i]<divisible)  
            continue;  
        if (números[i]%divisible==0)  
            divisibles.push(números[i]);  
    }  
    return divisibles;  
}
```

```
const números=[100,22,66,54,21], número=22;  
console.log(`De Los Números: ${números}`);  
console.log(`son divisibles por ${número}:  
${mostrarDivisibles(números,número)}`);
```

# [for]

El bucle *for* también puede incluir en el cuerpo del mismo la sentencia *break* , en este caso si se ejecuta finaliza la iteración en curso y el propio bucle.

El ejemplo de la derecha es un poco rebuscado y solo se emplea con el fin de enseñar la utilización de las sentencias *continue*, *continue* con etiqueta y *break*.

Muestra los múltiplos de 2, 3 y 5 comprendidos entre 1 y 100.

```
for (let i=0, j=2; i<100; i++)
{
    if(i%5!=0)
        continue;
    salta:
    for (j=2; j<4 ;j++)
    {
        if(i%j==0)
            if (j==3)
                console.log(`${i} es múltiplo de 2,3 y 5`);
            else
                continue salta
        else
            break;
    }
}
```

# [while]

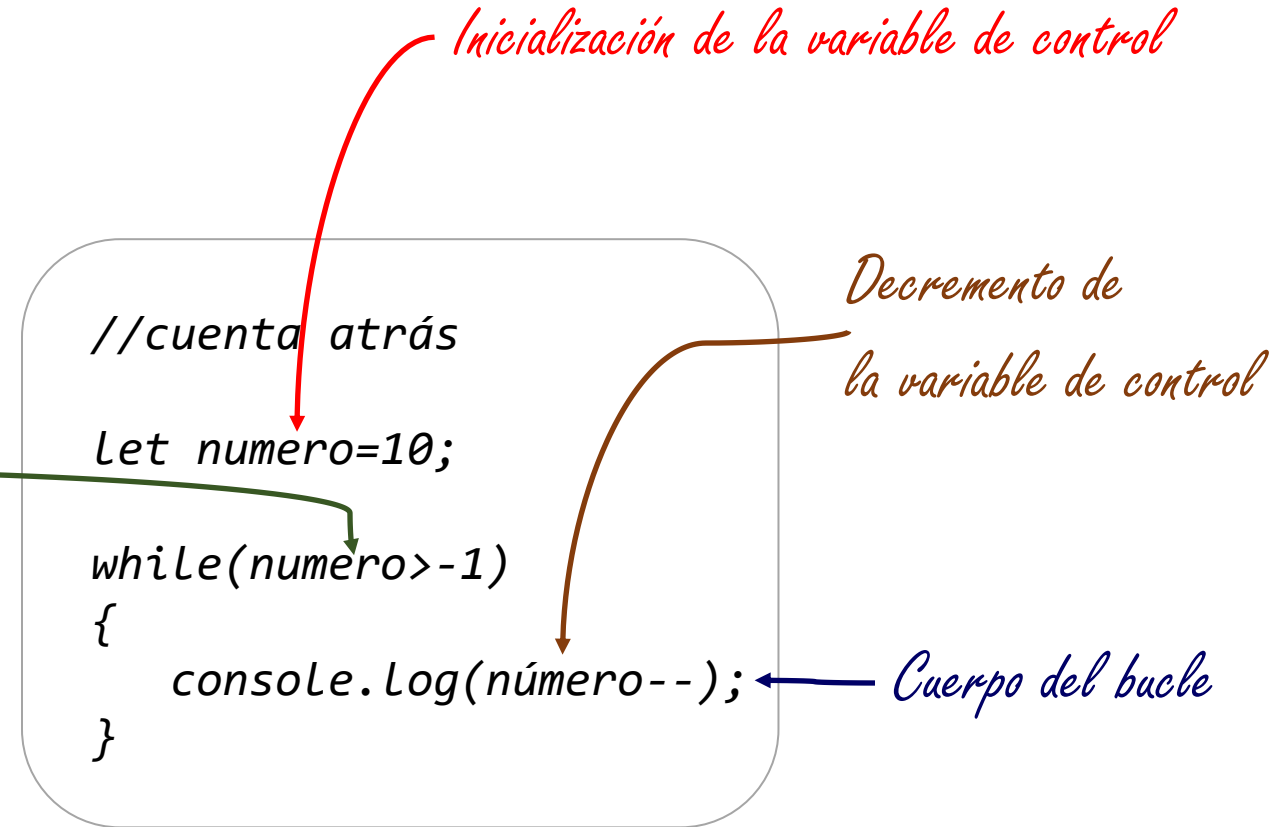
*while* (mientras) es otra sentencia utilizada para repetir un conjunto de sentencias mientras se cumpla una condición.

Su sintaxis es la siguiente:

```
while (condición)
{ }
```

El bucle comenzará comprobando la condición colocada cada entre paréntesis a continuación de la palabra reservada *while* si se cumple realizará una iteración, ejecutando el cuerpo del bucle, las sentencias colocadas entre llaves. Al finalizar la misma volverá a comprobar la condición, y si se cumple comenzará una nueva iteración.

*Condición de  
finalización del bucle*



# while

Ejemplo de bucle *while* en una función flecha que se le pasa una cadena de texto y escribe en consola la palabra al revés.

También podemos usar, de la misma forma que lo hacíamos con el bucle *for*, las sentencias *break* y *continue*. Su uso con el bucle *while* es exactamente el mismo.

```
const alReves=(palabra)=>{  
  let i=palabra.length, palabra2="";  
  
  while(palabra[--i])  
    palabra2+=palabra[i];  
  console.log(palabra2);  
}  
const miCadena="Jorge";  
alReves(miCadena);
```

egroJ



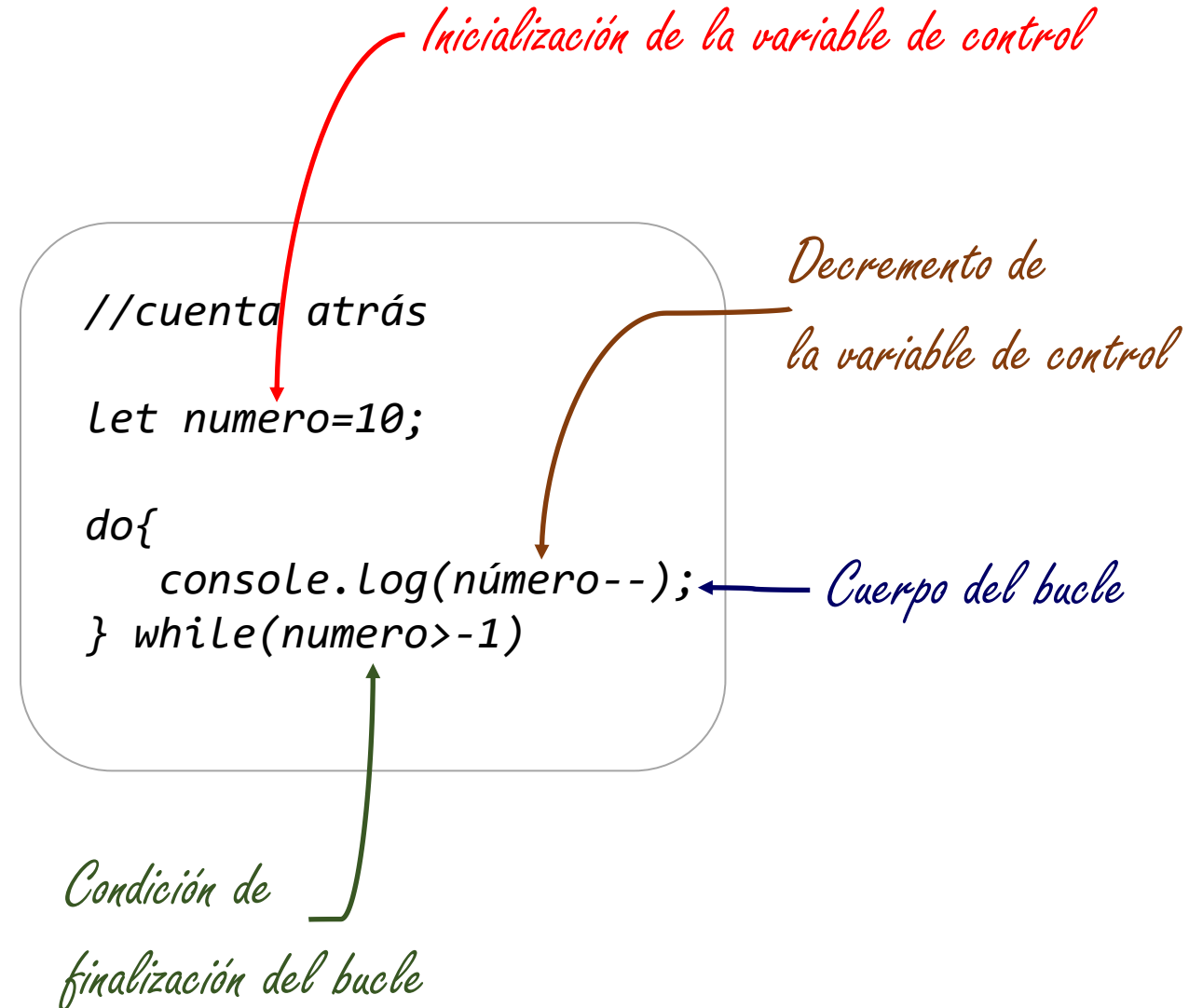
# [do-while]

*do-while* (hacer mientras) es otra sentencia, muy parecida a *while*, para repetir un conjunto de sentencias mientras se cumpla una condición.

Su sintaxis es la siguiente:

```
do
{ }
while (condición)
```

En este caso el bucle se va ejecutar por lo menos una vez ya que la comprobación de la condición, colocada cada entre paréntesis a continuación de la palabra reservada *while*, se comprueba al finalizar cada iteración y no al principio como en *while* y en el *for*.



## (do-while)

También se pueden usar las sentencias *break* y *continue*.

Ejemplo de bucle *do-while* de una función flecha que escribe el alfabeto en letras mayúsculas y minúsculas en la consola.

*String.fromCharCode()* es un método estático que devuelve una cadena de caracteres creada a partir de una secuencia de valores Unicode proporcionada como argumento.

Los métodos estáticos son llamados sin necesidad de instanciar una clase, crear el objeto de dicha clase.

En próximos temas se verán las clases

```
const alfabeto=()=>{  
  let i=65, lista="";  
  
  do{  
    if (i>90 && i<97)  
      continue;  
    lista += String.fromCharCode(i);  
    lista+= " ";  
  }while(++i<123)  
  console.log(lista);  
}
```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

# [for-of]

*for of* es una versión simplificada del bucle *for* muy utilizado con arrays, ya que simplifica mucho la iteración sobre el mismo.

Su sintaxis es la siguiente:

```
for (let elemento of ListaElementos)
{ }
```

En este caso, el bucle se ejecuta un número de veces igual al número de elementos presentes en *ListaElementos* (que puede ser un array, un *map*, un *set*). En cada iteración *elemento* va tomando cada uno de los valores almacenados en *ListaElementos*.

El bucle finaliza cuando ha recorrido todos los elementos en *ListaElementos*.

```
const meses=['Enero',  
            'Febrero', 'Marzo'];
```

```
for (let mes of meses){  
  console.log(mes);  
}
```

|         |
|---------|
| Enero   |
| Febrero |
| Marzo   |

*Cuerpo del bucle*

```
const meses=[{nombre:'Enero'},  
             {nombre:'Febrero'}];
```

```
for (let mes of meses){  
  console.log(mes);  
}
```

|                       |
|-----------------------|
| ► {nombre: 'Enero'}   |
| ► {nombre: 'Febrero'} |

## [for-in]

*for in* es otra versión del bucle *for*, utilizado con listas de elementos y parecido a *for of* pero con diferencias.

Su sintaxis es la siguiente:

```
for (let indice in ListaElementos)
{ }
```

El bucle se ejecuta un número de veces igual al número de elementos presentes en *ListaElementos*. En cada iteración *indice* va tomando los índices o claves de *ListaElementos* y no los valores almacenados en la lista.

El bucle finaliza cuando ha recorrido todos los elementos en *ListaElementos*.

```
const meses=['Enero',
'Febrero', 'Marzo'];
```

```
for (let indice in meses){
  console.log(indice);
}
```

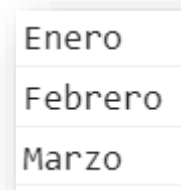


|   |
|---|
| 0 |
| 1 |
| 2 |

*Cuerpo del bucle*

```
const meses=['Enero',
'Febrero', 'Marzo'];
```

```
for (let indice in meses){
  console.log(meses[indice]);
}
```



|         |
|---------|
| Enero   |
| Febrero |
| Marzo   |

# [for-of vs for-in]

Las diferencias principales entre *for of* y *for in*, es que *for of* itera mejor sobre arrays y *for in* sobre objetos.

```
const coche={  
  marca: 'Seat',  
  modelo: 'Ibiza',  
  año: 2021,  
  precio: 12000  
};  
  
for (let propiedad in coche){  
  console.log(propiedad);  
}
```

|        |
|--------|
| marca  |
| modelo |
| año    |
| precio |

```
const coche={  
  marca: 'Seat',  
  modelo: 'Ibiza',  
  año: 2021,  
  precio: 12000  
};  
  
for (let propiedad of coche){  
  console.log(propiedad);  
}
```

✖ ▶ Uncaught TypeError: coche is not iterable  
at prueba.js:178

## [for-of vs for-in]

Para recorrer las propiedades y claves de un objeto con *for in* sería de la siguiente forma:

```
const coche={  
  marca: 'Seat',  
  modelo: 'Ibiza',  
  año: 2021,  
  precio: 12000  
};  
  
for (let propiedad in coche){  
  console.log(`${propiedad}=${coche[propiedad]}`);  
}
```

|              |
|--------------|
| marca=Seat   |
| modelo=Ibiza |
| año=2021     |
| precio=12000 |

## [for-of vs for-in]

Con *for of*, sobre objetos, se puede usar el método estático *Object.entries()* que devuelve un array formado por pares de claves y valores.

```
const coche={  
  marca: 'Seat',  
  modelo: 'Ibiza',  
  año: 2021,  
  precio: 12000  
};  
  
for (let [propiedad,valor] of Object.entries(coche)){  
  console.log(`${propiedad}=${valor}`);  
}
```

|              |
|--------------|
| marca=Seat   |
| modelo=Ibiza |
| año=2021     |
| precio=12000 |

# Bibliografía y recursos online

- [https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Grammar and types](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Grammar_and_types)
- <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/if...else>
- <https://tc39.es/ecma262/multipage/ecmascript-language-expressions.html#sec-assignment-operators>
- [https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Loops and iteration](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Loops_and_iteration)