null y undefined

UNDEFINED: Para Javascript, no existe. O bien no ha sido declarada o jamás se le asignó un valor.
NULL: Para Javascript, la variable existe. En algún momento, explícitamente, la variable se estableció a null.

Ejemplo:

La variable **nodef** está declarada pero no asignada, por tanto, javascript le asigna el tipo undefined.

A la variable **m** si se le asigna un valor aunque ya que la función match devuelve "null" cuando no encuentra coincidencias en la cadena. No hay "b" en la cadena "hola".

```
        var nodef;
        var str = "hola";

        var m = str.match(/b/);

        console.log(nodef); // undefined

        console.log(m); // null
```

Hoisting

En el siguiente enlace hay unos ejemplos sobre hoisting bastante buenos para entender cómo es el comportamiento que Javascript hace de las variables en el interior de las funciones, que es donde puede ocasionar ejecuciones inesperadas como se ve en el ejemplo. Lo puedes probar tú mismo y ver el resultado copiando y pegando el código de abajo.

https://www.etnassoft.com/2010/12/26/hoisting-en-javascript/

Puedes copiar y pegar este código en el VS code y probar a hacer click en el botón viendo el resultado del alert y el valor de x.

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Prueba Hoisting</title>
</head>

<body>
    <button>Prueba hoisting</button>
    <script type="text/javascript">
        var x = "Hello World";

        function foo(){
```

```
            alert( x ); // muestra undefined
            var x = 'New Value';
            alert( x ); // muestra 'New Value'
        }

        document.querySelector('button')
        .addEventListener('click', () => {foo()});
    </script>
</body>

</html>
```

Lo importante al final es lo que comenta en el ejemplo final:

La recomendación de los expertos es siempre declarar las variables locales al principio de su actual ámbito para evitar errores. De esta forma ganamos además legibilidad del código ya que agrupamos así todas nuestras variables en un mismo lugar permitiéndonos de un vistazo conocer el tipo de datos que esperan.

Objetos

¿Por qué utilizar objetos en lugar de simplemente varias variables?

Porque dan **estructura** al código, lo hace más fácil de programar y escalable (el código puede crecer en el futuro).

Se puede entender lo útiles que son con unos ejemplos:

Imagina que tienes que almacenar los datos de 5 productos y haces algo parecido a esto usando únicamente variables:

```
// Datos sobre el producto1
var id = 111;
var peso = 12;
var tieneDescuento = true;
var color = "#ff3";
var marca = "Asus";
var stock = 20;
// Datos sobre el producto2
var id2 = 222;
var peso2 = 12;
var tieneDescuento2 = false;
var color2 = "#ff4";
var marca2 = "Hp";
var stock2 = 10;
// Datos sobre el producto3
var id3 = 333;
var peso3 = 12;
var tieneDescuento3 = true;
var color3 = "#ff5";
var marca3 = "Logitech";
var stock3 = 0;
// Datos sobre el producto4
var id4 = 444;
var peso4 = 12;
var tieneDescuento4 = false;
var color4 = "#ff6";
var marca4 = "Dell";
var stock4 = 30;
// Datos sobre el producto5
var id5 = 555;
var peso5 = 12;
var tieneDescuento5 = false;
var color5 = "#ff7";
var marca5 = "Hp";
var stock5 = 0;
```

Perfecto, se puede hacer así. El problema es que ni siquiera estos datos están relacionados. Son datos “sueltos” que podrían corresponder a cualquier cosa.

De este modo se puede complicar mucho el código siendo más complejo y difícil de mantener.

Ahora el mismo ejemplo con objetos:

```
// Datos sobre los productos
var listaProductos = [
    {id: 111, peso: 12, tieneDescuento: true, color: "#ff3", marca: "Asus",stock: 20},
    {id: 222, peso: 12, tieneDescuento: false, color: "#ff4", marca: "Hp",stock: 10},
    {id: 333, peso: 12, tieneDescuento: true, color: "#ff5", marca: "Logitech",stock: 0},
    {id: 444, peso: 12, tieneDescuento: false, color: "#ff6", marca: "Dell",stock: 30},
    {id: 555, peso: 12, tieneDescuento: false, color: "#ff7", marca: "Hp",stock: 0}
];
```

A simple vista ya es mucho más ordenado, cada objeto tiene claras sus propiedades y te permite además utilizar las funcionalidades que tienen los objetos, del mismo modo que los strings tienen sus propiedades y funciones accesibles como slice(), replace(), length, toUperCase(), etc.

Además todos los productos tienen las mismas propiedades. Por ejemplo, su propiedad “peso” para todos los objetos es “peso” y no como en el ejemplo anterior (las variables -> peso, peso2, peso3, peso4 y peso5).

Ahora un ejemplo viendo la diferencia de manejar las variables y los objetos.

Queremos hacer un programilla que recorra todos los objetos y nos muestre en nuestra página aquellos que tengan existencias disponibles, es decir, los que tenga el stock mayor que 0.

Ejemplo con variables:

```
function getAvailableProducts() {
    var result = ``;
    if(stock>0) {
        result+= id + " " + peso + " " + tieneDescuento + " " + color + " " + marca + "<br>";
    }
    if(stock2>0) {
        result+= id2 + " " + peso2 + " " + tieneDescuento2 + " " + color2 + " " + marca2 + "<br>";
    }
    if(stock3>0) {
        result+= id3 + " " + peso3 + " " + tieneDescuento3 + " " + color3 + " " + marca3 + "<br>";
    }
    if(stock4>0) {
        result+= id4 + " " + peso4 + " " + tieneDescuento4 + " " + color4 + " " + marca4 + "<br>";
    }
    if(stock5>0) {
        result+= id5 + " " + peso5 + " " + tieneDescuento5 + " " + color5 + " " + marca5 + "<br>";
    }
    return result;
}
```

Ejemplo con objetos:

```
function getAvailableProducts2(listaProductos) {
    var result = ``;
    listaProductos.forEach(x => {
        if(x.stock>0) {
            result += x.id + " " + x.peso + " " + x.tieneDescuento + " " + x.color + " " + x.marca + "<br>";
        }
    });
    return result;
}
```

A mayor número de productos más notable sería la diferencia.

No hay que entender el código que he escrito de ejemplo, es solo para que te hagas una idea del desgaste que sería si no existieran las listas, los objetos y solo pudieras usar variables de tipos “simples” para cada dato sin relación entre ellos.

El resultado en ambos casos será el mismo:

111 12 true #ff3 Asus
222 12 false #ff4 Hp
444 12 false #ff6 Dell

Concatenación texto y números “+”, “$”

Se pueden crear string de 3 formas diferentes:

```
let cadena = "Una cadena de texto";
let cadenaDos = 'Segunda cadena de texto';
let cadenaTres = `Tercera cadena de texto`; // Permite multilinea
```

Se puede concatenar de estas 3 formas con el mismo resultado:

```
let result = cadena + cadenaDos;
let result2 = cadena.concat(cadenaDos);
let result3 = `${cadena} ${cadenaDos}`;
```

Ejemplo que podría causar errores:

```
let num=1, num2=2, res;
res = num + num2 + "hola" // -> “3hola”
res = `${num}${num2}hola` // -> “12hola”
res = num.toString().concat(num2+"hola") // -> “12hola”
```

Como se ve en el ejemplo, al concatenar de la forma simple mediante “+” podemos no obtener el resultado esperado si no sabemos qué tipos de datos se están manejando.

https://gyazo.com/fce18aec0697db661103f2afd4cc80e7

Resumen conversión automática de tipos primitivos (string y number)

```
> "100" - 57
< 43
> 100 - "57"
< 43
> "100" + 57
< "10057"
> 100 + "57"
< "10057"
> +"57"+100
< 157
```

La expresión "100" + 57 es ambigua porque combina un string con un number.

Si hay ambigüedad, JavaScript da prioridad al operador + de concatenación de strings, convirtiendo 57 a string y concatenando ambos strings.

La expresión +"57" también necesita conversión automática de tipos

El operador + solo está definido para el tipo number (no hay ambigüedad) y convierte automáticamente en number.

Ejemplo console.log (crear objeto "consola" con función "log")

miConsola es un objeto que he creado yo para que te hagas una idea de cómo está construido el objeto console con su función log(). No es importante que entiendas el código, es solo un ejemplo funcional. No tiene sentido programar tal cosa puesto que ya existe console.log y además lo utilizo para que funcione miConsola.

```
// Creo el objeto miConsola
const miConsola = {
    log: (...args) => {
        for(let i=0; i<args.length; i++){
            console.log(args[i])
        }
    }
}
miConsola.log("hola", "que", "tal");
```

Ejercicios (se puede copiar y pegar)

```
var producto = {
            id: 1111,
            precio: 20,
            tieneDescuento: true,
            otrasProps: {
                color: "negro",
                marca: "Asus",
                modelo: "H-777"
            },
            stock: 20
        };

        // Añadir al producto la propiedad description
        // que tendrá como valor la concatenación del id y el peso.
        // y mostrar por consola la description del producto.
        // resultado esperado por consola // 111120

        // Añadir una propiedad nueva (material = "aluminio") dentro de o
trasProps y mostrarla por consola

        // Modificar el valor "negro" de la propiedad color por "blanco"
y mostrarla por consola
```

```
        // Guardar en una variable el objeto otrasProps y acceder a la pr
opiedad marca desde esa variable
```

Por cierto, además del “.”para acceder a la propiedad de un objeto también puedes usar esta sintaxis:

```
producto[“precio”] // → 20
```

```
producto[“otrasProps”][“color”] // → “negro”
```

Y también puedes combinarla:

```
Producto[“otrasProps”].color // → “negro”
```

Por si lo ves por ahí en algún ejemplo, que no te parezca raro.

Recordatorio Comparaciones

=== // mismo valor y tipo

!== // no el mismo valor ni el mismo tipo

> // mayor que

< // menor que

>= // mayor o igual

<= // menor o igual

&& // and

|| // or

! // not, Niega la expresión

IF – ELSE

```
SI ( condición1 ) {
      // Código que se ejecuta si condición1 es verdadera
} SINO {
      // Código que se ejecuta si condición1 es falsa.
}
```

IF – ELSEIF – ELSE

```
SI ( condición1 ) {
      // Código que se ejecuta si condición1 es verdadera
} SINO SI ( condición2 ) {
      // Código que se ejecuta si condición1 es falsa y condición2 es verdadera
} SINO {
      // Código que se ejecuta si condición1 es falsa y condición2 es falsa
}
```

SWITCH

```
BUSCA_CASO_COINCIDENCIA_DE_LA EXPRESIÓN ( expresión ) {
// expresión devolverá un valor, ej num = 1
      CASO1: CASO1
            // Código que se ejecuta si el valor de la expresión coincide con el del CASO1
            (break) SALGO DEL BLOQUE SWITCH
      CASO2: CASO2
            // Código que se ejecuta si el valor de la expresión NO coincide con el del CASO1 y SÍ coincide con CASO2
            (break) Y SALGO DEL BLOQUE SWITCH
      CASO_POR_DEFECTO:
            // Código que se ejecuta cuando no se cumpla ninguno de los casos
}
```