HINTS

PALABRA CLAVE “BREAK”

“Break” se utiliza comúnmente en el **switch**, pero también en otros ciclos que nos permiten controlar el flujo del código. A continuación, vemos dónde se puede ubicar en un **switch**:

```
    switch (valor) {
        case '1':
            console.log(`Para la opción '${valor}' se ejecuta este
código.`);
            break;
        default:
            break;
    }
```

Esta palabra clave, en español se traduciría a *quebrar o romper*; y se refiere a romper la ejecución del código, actuando como un freno, y también se puede emplear para detener la ejecución del código.

PALABRA CLAVE “CONTINUE”

“continue”, que en castellano podemos traducir a *continuar*, termina la ejecución de las instrucciones en la iteración actual de un ciclo, y continúa la ejecución del ciclo con la siguiente iteración. Por ejemplo, considerando la siguiente función:

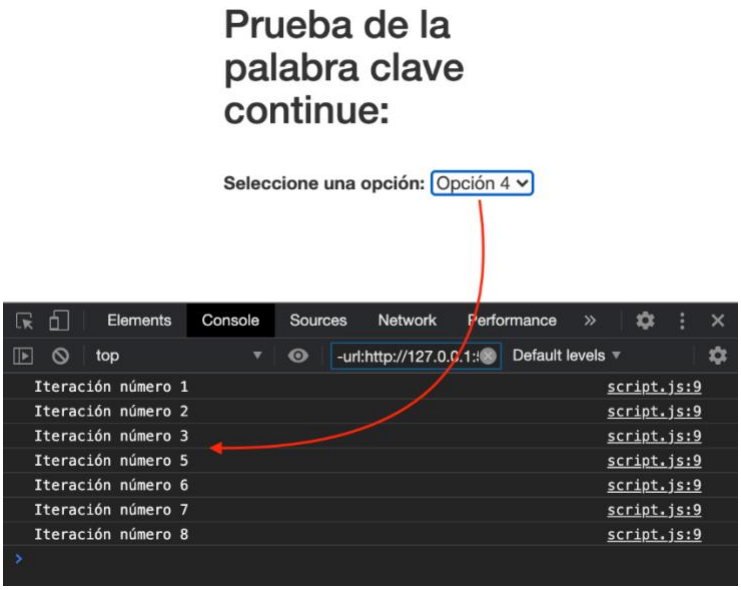
```
    <form>
        <h2> Prueba de la palabra clave continue:</h2><br>
        <label for="opciones">Seleccione una opción:</label>
        <select name="opciones" onchange="cambia(value)">
            <option value="1">Opción 1</option>
            <option value="2">Opción 2</option>
            <option value="3">Opción 3</option>
            <option value="4">Opción 4</option>
            <option value="5">Opción 5</option>
            <option value="6">Opción 6</option>
            <option value="7">Opción 7</option>
        </select>
    </form>
```

```
function cambia(valor) {

    for (let i = 1; i < 9; i++) {
        if (i === Number(valor)) {
            continue;
```

```
        }
        console.log(`Iteración número ${i}`)
    }
}
```

En esta función establecemos que, si dentro de la interacción, la variable `i` alcanza a tener el valor que ingresamos mediante el HTML, a través de la palabra clave `continue` la saltamos, y continuamos con el resto del código. En nuestro navegador, el resultado será el siguiente:



Si podemos notar, la opción que seleccionamos es justamente la que queríamos que la iteración se sentara, y mediante la consola, nos damos cuenta de que eso es justamente lo que sucedió, lo que nos ayuda a comprender como funciona esta palabra clave dentro de una iteración.

OPERADORES DE INCREMENTO Y DECREMENTO

Existen muchos operadores interesantes, que están disponibles para usar en JavaScript. Dos operadores simples, pero importantes, que podemos implementar en nuestros desarrollos, son los de incremento y decremento. El primero, se escribe como dos signos más, después del operando `(x++)`; mientras que el segundo se escribe como dos signos de resta, después del operando `(y--)`.

El operador de incremento, aumenta en uno el valor de su operando; mientras que el de decremento, resta uno. Revisemos un ejemplo que comprueba este hecho.

```
1  // Incremento
2  let x = 3;
3  console.log("valor original (x): " + x)
4
5  x++;
6  console.log("valor incremento (x++): " + x)
7
8  // Decremento
9  let y = 3;
10 console.log("valor original (y): " + y)
11
12 y--;
13 console.log("valor decremento (y--): " + y)
```

Si ahora nos dirigimos a nuestra consola, podremos ver que los operadores efectivamente suman y le restan el valor de 1 a sus operandos, respectivamente.

```
valor original (x): 3                    javascript.js:3
valor incremento (x++): 4                javascript.js:6
valor original (y): 3                    javascript.js:10
valor decremento (y--): 2                javascript.js:13
>
```

Estos operadores son claves al momento de usar el ciclo for, pero recomendamos que analices circunstancias en las que puedes utilizar este operando para simplificar tus desarrollos.

PRECEDENCIA DE OPERADORES

Ahora consideraremos un tema más teórico, el cual es importante tener en cuenta al implementar muchas operaciones: la precedencia de los operadores. Este concepto se refiere a la prioridad otorgada a los operadores, al analizar una declaración que tiene más de uno realizando operaciones en ella. Es importante asegurar el resultado correcto, y también ayudar al compilador a comprender cuál debe ser el orden de dichas operaciones.

Esto es muy similar a cuando se intenta realizar un cálculo complejo en una calculadora científica, si no le indicamos las instrucciones correctas, no tendremos el resultado esperado; y lo mismo sucede al realizar operaciones en JavaScript, y en cualquier otro lenguaje de programación.

Los operadores con prioridades más altas se resuelven primero. Pero, a medida que se avanza en la lista, disminuye la prioridad y por ende su resolución.

Ahora, en una nota similar, la asociatividad marca el polo opuesto de lo que es la precedencia. La asociatividad en general establece que, independientemente del orden de los operandos para una operación dada, el resultado sigue siendo el mismo. La precedencia se usa para decirle al compilador qué operaciones se deben efectuar primero. Por ejemplo, considerando las siguientes:

```
1 // Asociativo
2 console.log((2 + 3) + 4)
3 console.log(2 + (3 + 4))
4
5 // Precedencia
6 console.log(2 >= 3)
7 console.log(1 != 4)
```

Resultará así:

```
9                                    javascript.js:2
9                                    javascript.js:3
false                                javascript.js:6
true                                 javascript.js:7
>
```

Las primeras dos operaciones son asociativas, ya que no importa su orden. Desde el tercer caso en adelante es la precedencia, donde para alcanzar el resultado deseado, tiene que haber un orden adecuado en el que se realizarán las operaciones.

Como se puede observar, este concepto no proviene de las reglas de programación, sino que es parte de los fundamentos del álgebra. Si usamos este conocimiento a nuestro favor, seremos más capaces de escribir y depurar código con múltiples operaciones.

OPERADORES DE ASIGNACIÓN COMPUESTOS

Hasta el momento, hemos trabajado con el operador de asignación igual (=), con el que asignamos un valor a una variable.

```
1 var numero = 1;
```

Y si queremos sumarle un valor (o restarle, dividirle o multiplicarle), realizamos la siguiente acción:

```
1 numero = 1 + 2;
```

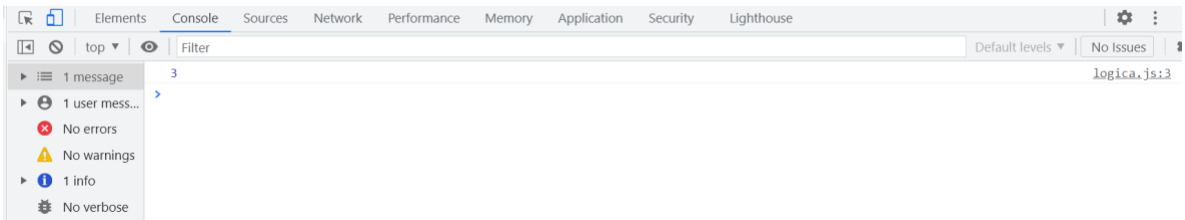
Sin embargo, existen los operadores de asignación compuestos, que utilizan el operador de asignación combinado con un operador aritmético, y de esta forma abrevia o reduce ciertas expresiones. Con esto podemos, al mismo tiempo, realizar la operación y la asignación.

```
1 numero += 2;
```

Imprimamos ambas posibilidades.

```
1 var numero = 1;
2 numero = 1 + 2;
3 console.log(numero);
```

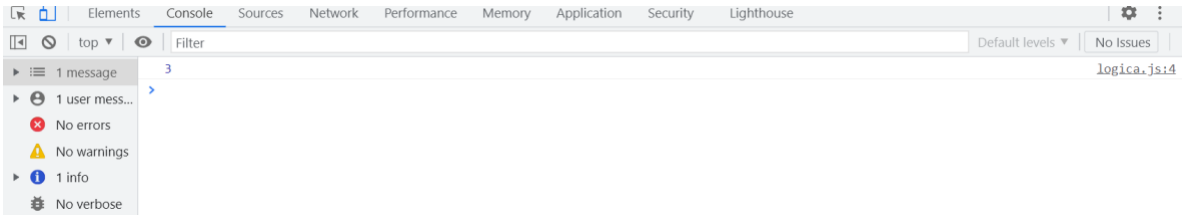
Nos muestra lo siguiente:



Ahora, imprimamos la segunda posibilidad.

```
var numero = 1;
//numero = 1 + 2;
numero += 2;
console.log(numero);
```

Dando como resultado:



Los operadores de asignación compuestos son:

Operador	Ejemplo	Expresión equivalente
+=	a += b;	a = a + b
-=	a -= b;	a = a - b
*=	a *= b;	a = a * b
/=	a /= b;	a = a / b
%=	a %= b;	a = a % b

ELSE IF

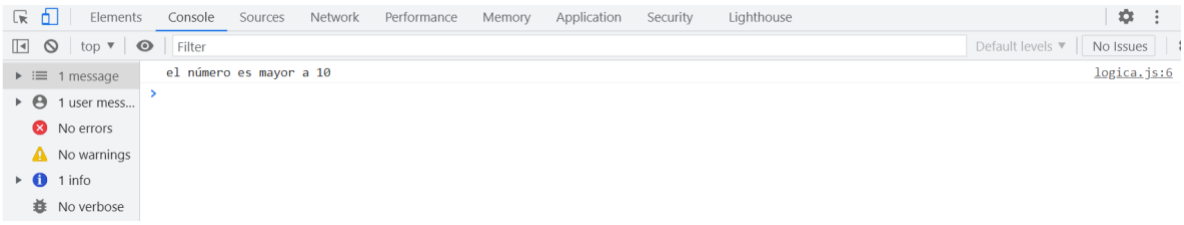
Al trabajar con la condicional **if**, posterior a la evaluación realizada, en ocasiones requeriremos realizar otra evaluación. Para eso existe la condición **else if** (entonces sí), que se utiliza de la siguiente manera:

```
var numero = 15;

if (numero < 2) {
    console.log("el número menor a dos");
} else if (numero > 10) {
    console.log("el número es mayor a 10");
```

```
} else {
    console.log("el número es distinto");
}
```

De esta forma, el flujo ingresa al `if`, y como no cumple la condición, pasa a la siguiente evaluación `else if`. Al cumplirla ingresa allí; y en caso de que no, ingresaría al `else`.



CICLO WHILE

En este CUE aprendimos sobre el bucle `do while`, que es una variante del bucle `while`.

El ciclo `while` ejecutará el bloque de código en su interior, una vez antes de verificar si la condición es verdadera, y luego repetirá el ciclo.

Esta es la sintaxis de un bucle `while`:

```
while (condicion) {
  // codigo por ejecutar
}
```

Como podemos ver, el ciclo while es muy similar a un `for`, con la excepción de que este último requiere definir una variable de incremento dentro de sus argumentos.

Veamos un ejemplo de los ciclos `while`. Crearemos una cuenta regresiva, la cual mostraremos en un HTML.

```
<body>

    <p id="cuenta"></p>

    <script>
        let cuentaRegresiva = "";
```

```
 7          let i = 3;
 8          while (i > 0) {
 9              cuentaRegresiva += "<br>En " + i;
10              i--;
11          }
12          document.getElementById("cuenta").innerHTML = cuentaRegresiva;
13      </script>
14  </body>
```

En este ejemplo, el ciclo `while` ejecuta una vez las instrucciones en las líneas 9 y 10, pues no verifica si la condición es cierta. Luego, procede a indicar los siguientes dos valores de nuestra variable `i`, como vemos en el navegador:

En 3
En 2
En 1

CONDICIONES DE BORDE

Son restricciones necesarias para la solución de un problema, en donde ésta debe cumplir un requisito. A estos tipos de problema, también se les llama condiciones de borde. Corresponden a un tema relevante a la hora de usar ciclos.

Por ejemplo, ¿qué pasa si una aplicación requiere que un usuario ingrese un número par y, sin embargo, el usuario solo ingresa un número impar?, ¿Debería dejar de funcionar la aplicación, o debería no hacer nada?: aquí es donde se deben utilizar las condiciones de borde, para que la aplicación pueda dar retroalimentación al usuario, y así cumplir su objetivo.

Primero preparamos nuestro HTML.

```
1  <div class="form-group">
2
3              <label for="exampleFormControlInput1">Ingresa un número par que
4  no sea 0</label>
5              <input type="number" class="form-control" id="num">
6          <button type="click" class="btn btn-primary"
7  onclick="validarNum()">Enviar</button>
8  </div>
```

Esto nos da el siguiente formulario:

Después, podemos definir nuestras condiciones de borde. En este caso, son las siguientes:

- Debe ser par.
- No puede ser cero.
- No puede ser impar.

Hay muchas formas de implementarlas, pero una de las más sencillas de hacerlo, es mediante condiciones `if`; las cuales están definidas en nuestra función `validarNum`, que se ejecuta cada vez que el usuario hace clic en el botón “enviar”.

```
function validarNum() {
    var numero = document.getElementById("num").value;
    console.log(numero)

    if (numero%2 == 0) {
        alert("Excelente")
    }
    if (numero%2 !== 0) {
        alert("Tiene que ser un número par")
    }
    if (numero = 0) {
        alert("No puede ser cero!!")
    }
}
```

Te recomendamos que pruebes este código en tu navegador, para estudiar el uso práctico de diseñar código con condiciones de borde.

CICLOS ANIDADOS

Cuando tenemos un bucle que se ejecuta dentro de otro, lo llamamos bucle o ciclo *anidado*. Podemos utilizarlos para realizar operaciones complejas, de forma sencilla. Veamos esto en un ejemplo, donde primero crearemos un pequeño programa que dibuje cuadrados del símbolo #.

Para hacerlo, agregaremos un elemento `h1` a nuestro HTML:

```
    <h1 id="miCuadrado"></h1>
```

Luego, incluiremos lo siguiente en nuestro Script:

```
var filas = window.prompt("Ingresa cantidad de filas");
var columnas = window.prompt("Ingresa cantidad de columnas");

for (i = 0; i < filas; i++) {
    for (let j = 0; j < columnas; j++) {
        document.getElementById("miCuadrado").innerHTML += "#"
    }

    document.getElementById("miCuadrado").innerHTML += "<br>"
}
```

En primer lugar, le pediremos al usuario que ingrese la cantidad de filas y columnas que desea que tenga el cuadrado, y luego usaremos bucles for anidados para dibujar el cuadrado o el rectángulo.

El bucle `for` más interno dibuja las columnas, y el exterior se encarga de las filas, y produce el salto de línea.

Si intentamos dibujar un cuadrado de 5x5, veremos el siguiente resultado en nuestro navegador:

Este es solo un ejemplo de cómo podemos usar bucles anidados para unificar procesos, y simplificar funciones de varias capas.

COMBINACION DE CICLOS CON INSTRUCCIONES IF/ELSE

Al usar bucles también podemos crear funcionalidades complejas, combinándolas con sentencias if/else. Por ejemplo, pensemos que estamos trabajando en una aplicación para un almacén, donde necesitamos categorizar los alimentos. Usando un bucle for, podemos iterar sobre los elementos de una matriz con todos los números de código de barras de productos alimenticios, y usando una declaración if/else, podemos categorizar a qué corresponden dichos códigos de barras.

Si estos códigos de barras incluyen un 0 y un 6 de forma consecutiva, entonces ese producto corresponde a un alimento sin gluten, y si no, solo es un producto alimenticio normal.

Podemos programar esta funcionalidad combinando un bucle for con una declaración if/else, de la siguiente manera:

```
        var cod_barra_productos = [7102, 4556, 1949, 1706, 8930, 4971,
5157, 7696, 5306, 9210, 6317, 2551, 4202, 7298, 6281, 4582, 2518, 4720,
5276, 5643, 2736, 2886, 7968, 9838, 6305, 5371, 7338, 9285, 5462, 4569,
8764, 8863, 2708, 2281, 3892, 7810, 4429, 7070, 6266, 5064, 8176, 3814];
        var cant_comida_sin_gluten = 0;
```

```
        var cant_comida_normal = 0;

        for (let i = 0; i < cod_barra_productos.length; i++) {

            if (cod_barra_productos[i].toString().includes("06")) {
                cant_comida_sin_gluten += 1;
            } else {
                cant_comida_normal += 1;
            }
        }

        console.log("Nuestro almacen tiene " + cant_comida_sin_gluten + "
productos sin gluten.")
        console.log("Nuestro almacen tiene " + cant_comida_normal + "
productos con gluten.")
```

Luego de ejecutar nuestro código, podemos ver en nuestra consola el siguiente resultado:

```
Nuestro almacen tiene 3 productos sin gluten.        index.html:26
Nuestro almacen tiene 39 productos con gluten.       index.html:27
>
```