UT03.1: Tutorial y resumen JavaScript

Partimos de la web de ejemplo propuesta. Esta web está diseñada utilizando el framework ***Boostrap v4*** (https://getbootstrap.com/) para hacerla responsive (adaptativa a cualquier tamaño de dispositivo).

Para instalar el framework en la página hemos utilizado CDN (Red de distribución de contenidos en español). En la cabecera de la página importamos las plantillas CSS utilizadas por el framework.

```
    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/boots
trap/4.5.2/css/bootstrap.min.css"
        integrity="sha384-
JcKb8q3iqJ61gNV9KGb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN5t9UJ0Z" crossor
igin="anonymous">
    <!-- Google Fonts -->
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400
,600,700" rel="stylesheet">
    <!-- Font Awesome -->
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.
3.1/css/all.css"
        integrity="sha384-
mzrmE5qonljUremFsqc01SB46JvROS7bZs3IO2EmfFsd15uHvIt+Y8vEf7N7fWAU" crossor
igin="anonymous">
```

En la parte final de la página importamos las librerías JavaScript con las funciones de *Bootstrap* y los frameworks que necesita para su funcionamiento, *jQuery* y *Popper*.

```
    <!-- jQuery first, then Popper.js, then Bootstrap JS -->
    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
        integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
        crossorigin="anonymous"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/
umd/popper.min.js"
        integrity="sha384-
ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49"
        crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bo
otstrap.min.js"
        integrity="sha384-
B4gt1jrGC7Jh4AgTPSdUtOBvfO8shuf57BaghqFfPlYxofvL8/KUEfYiJOMMV+rV"
        crossorigin="anonymous"></script>
```

1. Inicio

Creamos una carpeta denominada *js* para incluir dos ficheros que nos permita introducir el código que necesitamos en la página. La idea de tener dos ficheros independientes es separar el código que puede ser reutilizable entre páginas, del código que depende de la página que estamos desarrollando.

- **util.js**: Contendrá el código reutilizable.
- **home.js**: Contendrá el código que depende de la estructura de la página actual.

El siguiente código muestra la importación de código externo, justo después de las librerías de *Bootstrap*.

```
    <script src="js/util.js"></script>
    <script src="js/home.js"></script>
```

Tenemos que observar que la forma correcta de hacerlo es utilizando un tag de cierre. El siguiente código muestra como sería hacerlo de forma errónea utilizando un tag con el cierre embebido. Podemos ver como el editor marca el error en rojo.

```
<script src="formaIncorrecta.js"/>
```

1.1. Función para mostrar mensajes

Una función es un bloque de código que puede ser invocado mediante un nombre, y pasándole una serie de parámetros que nos permita personalizar la ejecución.

Creamos una función *showMessage* que mostrará un texto en un elemento concreto de nuestra página en *util.js* ya que será reutilizable.

```
function showMessage (message){
    document.getElementById("message").innerHTML = message;
}
```

Como vemos estamos utilizando el API de DOM de la página para acceder al contenido de un elemento cuyo identificador es *message*.

Elegimos un elemento para añadir este identificador, por ejemplo, en la zona de descuento de la cabecera.

```
<h1 id="message" class="col-sm-12">GET A GRIP</h1>
```

En el fichero *home.js* podemos realizar la invocación de la función.

```
showMessage ("Mensaje de prueba");
```

1.1.1. Funciones con parámetros múltiples

Crea una nueva función donde pases el identificador del elemento y el mensaje como parámetro e invócala.

```
function showMessageInElement (id, message){
    document.getElementById(id).innerHTML = message;
}
```

Hacemos la invocación desde *home.js*.

```
showMessageInElement ("message", "Mensaje de prueba2");
```

1.2. Formateando Código

Algunas convenciones que debemos utilizar para mejorar la legibilidad del código, y que son comúnmente utilizadas son:

- Variables de nombre con estilo conocido como *camelCase*. Comenzamos con minúscula y cada palabra de la variable la ponemos en mayúscula. Ejemplo "showMessageInElement". También deben comenzar por letra.
- Constantes en mayúsculas.
- Nombrado ficheros externos en minúsculas.
- Espacios alrededor de operadores.

```
let x = y + z;
let values = ["Volvo", "Saab", "Fiat"];
```

- Sangría de código con dos espacios.
- Reglas de declaraciones simples.
    - Finalizar siempre una instrucción simple con un punto y coma.
- Reglas de declaraciones de bloque.
    - Colocar el soporte de apertura al final de la primera línea.
    - Utilizar un espacio antes del soporte de apertura.
    - Colocar el soporte de cierre en una nueva línea, sin espacios iniciales.
    - No terminar una instrucción compleja con un punto y coma.

```
function toCelsius(fahrenheit) {
  return (5 / 9) * (fahrenheit - 32);
}
```

- Longitud de línea de 80 caracteres como máximo.

1.3. Comentarios

Dos tipos de comentarios

- En línea: //
- De múltiples líneas: /* */

```
// Comentario de línea
/*
function toCelsius(fahrenheit) {
  return (5 / 9) * (fahrenheit - 32);
}
*/
```

2. Variables y constantes

Tenemos tres tipos primitivos en JavaScript:

- number

- string
- boolean

La palabra reservada `let`.

```
let price = 13.99;
let name = "Portátil SSD";
let discounted = true;
```

2.1. Declaración de variables

Aunque hay otra forma de crear variables, es recomendable el uso de `let` ya que delimita el ámbito de aplicación de la variable al entorno de su declaración.

```
let welcome = "Hola mundo";
showMessage (welcome);
```

Ejercicio

Utilizar las variables anteriores para cambiar texto en otros objetos.

2.2. Nombrado de variables

Seguimos dos reglas básicas:

Empezar con uno o más:	_ $ letra	n nombre nombre_12
Seguido de 0 o más:	_ $ Letra número	nombrePersona _nombrePersona $nombrePersona _123 __proto__

2.3. Asignación de valores a variables

JavaScript es un lenguaje débilmente tipado, por lo que en la declaración no es necesario indica el tipo de dato, sino que el tipo es inferido en función del valor de la variable.

Podemos cambiar el tipo de una variable tan solo asignando un valor con diferente tipo a la variable y el intérprete no genera error.

```
let welcome = "Hola mundo";
welcome = 14;
showMessage (welcome);
```

2.4. Uso de constantes

La palabra reservada `const` la utilizamos para crear constantes. Lo habitual es hacerlo al comienzo del bloque de código donde utilizaremos la constante. Debemos tener en cuenta que en caso de reasignación, el intérprete generará un error. Por acuerdo, el nombre de la constante la debemos definir en mayúsculas.

Asignamos un nuevo identificador denominado *discount* en el HTML de la página, en la siguiente cabecera.

```
<h2 id="discount">20% OFF</h2>
```

Modificamos la cabecera desde *home.js*.

```
const IVA = 21;
showMessageInElement ("discount", IVA);
```

2.5. La palabra reservada var

Con la instrucción `var` podemos declarar variable que sobrepasen el ámbito de la declaración. El proceso de declaración de este tipo de variables es denominado **hoisting**, por el cual el intérprete hace dos pasadas al código. Por un lado, el intérprete busca las declaraciones de variables con `var`, y en la segunda pasada se ejecutan las asignaciones.

El código siguiente ejecuta primero la declaración de la variable *total*, para que en la segunda pasada se ejecute la invocación de la función *showMessage*. En ese momento *total* no tiene ningún valor, por eso está *undifined*. El código continua y se le asigna un valor a *total* y a partir de ese punto podemos ver su valor por la consola.

```
showMessage (total);

var total = 50.55;
console.log (total);
```

Si modificamos `var` por `let`, vemos que el intérprete genera un error ya que la variable no está definida.

2.6. Operador coma

Permite declarar más de una variable a la vez, pero no asigna valores.

```
let var1, var2, var3;
```

Para asignar valores iguales tendríamos que utilizar el operador de asignación (=).

```
let var1, var2, var3;
let var4 = var5 = var6 = 123;
console.log (var4);
console.log (var5);
console.log (var6);
```

3. Tipos y operadores

En el siguiente enlace de MDN podemos consultar la referencia de operadores en la versión ES6. Veamos los más básicos.

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators

Creamos un nuevo fichero externo denominado ***03_operators.js*** para realizar los ejemplos.

```
<script src="./js/util.js"></script>
<script src="./js/home.js"></script>
<script src="./js/03_operators.js"></script>
```

3.1. Numbers

El operador typeof nos devuelve el tipo de variable que estamos utilizando.

```
showMessage (typeof price);
```

3.1.1. Operadores aritméticos

Al igual que en la mayoría de los lenguajes, los operadores aritméticos estándar son la suma (+), la resta (-), la multiplicación (*), la división (/) y el residuo (%).

Un ejemplo de ejecución sería el siguiente:

```
price = 15;
showMessage (price); //15
price = price + 12;
showMessage (price); //27
price = price - 7;
showMessage (price); //20
price = price * 2;
showMessage (price); //40
price = price / 5;
showMessage (price); //8
price = price % 3;
showMessage (price); //2
```

Un operador que ha sido añadido a ES6 es la potencia (**). Veamos un ejemplo de su uso.

```
price = price ** 3;
showMessage (price); //8
```

3.1.2. Operadores de asignación

Los operadores de asignación asignan valores al operando izquierdo en función del derecho. Además del operador simple, tenemos operadores compuestos que son una abreviación que permite reutilizar el operando izquierdo de forma implícita con el operador. La siguiente tabla muestra los principales operadores de asignación compuestos.

Nombre	Operador abreviado	Significado
Asignación de adición	x += y	x = x + y
Asignación de resta	x -= y	x = x - y
Asignación de multiplicación	x *= y	x = x * y
Asignación de división	x /= y	x = x / y
Asignación de residuo	x %= y	x = x % y

Tabla 1 Operadores de asignación compuestos

El siguiente código es similar al ejemplo anterior, pero utilizando operadores compuestos.

```
price = 15;
showMessage (price); //15
price += 12;
showMessage (price); //27
price -= 7;
showMessage (price); //20
```

```
price *= 2;
showMessage (price); //40
price /= 5;
showMessage (price); //8
price %= 3;
showMessage (price); //2
```

3.1.3. Operadores unarios

Podemos incrementar (++) y decrementar (--) en 1 el valor de una variable.

```
showMessage (++price); //3
showMessage (--price); //2
```

Estos operadores hacen un *pre-incremento* o *pre-decremento*. También podemos ejecutar el operador posteriormente al uso de la variable.

```
showMessage (price++); //2
console.log (price); //3
showMessage (price--); //3
console.log (price); //2
```

En este tipo de operadores de *post-incremento* o *post-decremento*, primero accedemos a la variable, y posteriormente se cambia su valor.

Tenemos otros dos operadores unarios:

- **Negación unaria (-)**: Devuelve la negación del operando.
- **Positivo unario (+)**: Convierte el operando en un *number*.

```
showMessage (-price); //-2
showMessage (typeof +"123"); //number
```

3.1.4. Precedencia de operadores

En el siguiente enlace podemos encontrar la precedencia de ejecución de operadores en una expresión.

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Operator_Precedence#Table

En la tabla podemos ver:

- **El nivel de precedencia**. A menor número mayor precedencia en su ejecución, siendo el paréntesis que agrupa un resultado el de mayor precedencia, y por tanto, el que tenemos que resolver primero.
- **Tipo de operador**.
- **Asociatividad**. Indica cómo se aplica el operador, de izquierda a derecha o de derecha a izquierda.
- **Operador**.

```
price = 3 + 2 * 2;
showMessage (price); //7
price = (3 + 2) * 2;
showMessage (price); //10
```

3.1.5. Precisión

El intérprete de JavaScript no destaca por su precisión a la hora de trabajar con valores en coma flotante. En el siguiente ejemplo vemos como podemos trabajar con números en coma flotante.

```
// Precisión
let taxRate = 0.21;
showMessage (price * taxRate); //2.1
```

Hay ocasiones donde el intérprete pierde esta precisión. Veamos un ejemplo.

```
price = 1.1 + 1.3;
showMessage (price); //2.4000000000000004
```

No conviene realizar operaciones en coma flotante con el intérprete de JavaScript por esta razón. Una solución sería aplicando redondeos.

```
showMessage (typeof +price.toFixed(2)); //2.40
```

3.1.6. Números negativos

Utilizamos el operador unario de negación. Este operador multiplica por -1 el operando, que en este caso es negativo, por lo que aplica un valor positivo por la propiedad de multiplicar dos números negativos, produce un valor positivo.

En el ejemplo, la primera expresión muestra como el operador de negación multiplica por -1 el operando, que en este caso es negativo, por lo que aplica un valor positivo.

```
price = 20 - -2;
showMessage (price); //22
```

3.2. String

Para declarar un *string* podemos utilizar tanto comilla doble como comilla simple.

```
let message = "Hola Mundo Doble";
showMessage (message);
message = 'Hola mundo Simple';
showMessage (message);
```

Si necesitamos las añadir la comilla doble o la comilla simple a nuestro *string*, el intérprete pensaría que estamos rompiendo la declaración, por lo que tenemos que utilizar el carácter de escape que para JavaScript es la barra invertida (\). El carácter de escape hace que el siguiente carácter de la cadena no se procese con la funcionalidad que tiene asignada, y se utilice con un carácter simple.

```
message = "Hola \"Mundo\"";
showMessage (message);
message = 'Hola \'Mundo\'';
showMessage (message);
```

En el siguiente enlace se puede consultar la notación de caracteres de escape en MDN.

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/String

Por último, podemos crear un *string* utilizando comillas invertidas, que además nos ofrece la posibilidad de obtener el valor de una variable utilizando la notación `${variable}`. Veamos un ejemplo.

```
let username = 'Pablo';
message = `Hello ${username}`;
showMessage (message);
```

Las comillas inversas nos dan la opción de formatear saltos de línea sin necesidad de utilizar el carácter de escape. Debemos recordad que en HTML se eliminan no podemos formatear saltos de línea mediante caracteres.

```
message = `Mantener
saltos
de línea
sin carácter
de escape.
`;
console.log (message);
```

3.2.1. Manipular String

El operador concatenación (+) permite unir dos *strings*.

```
showMessage ("Hola " + "mundo!");
```

Una variable *string* realmente es un objeto, con un conjunto de métodos y de propiedades. Podemos invocar sus métodos y acceder a sus propiedades con la notación típica de objetos. Veamos unos ejemplos. El siguiente código muestra el contenido de la variable *message* en minúsculas y obteniendo la subcadena desde el carácter de la primera posición respectivamente.

```
message = 'Hola Mundo';
showMessage (message.toLocaleLowerCase());
showMessage (message.substring(1));
```

En cuanto a las propiedades tenemos *length* que nos devuelve la longitud de la cadena.

```
showMessage (message.length); //10
```

Nuevamente la referencia de *string* está disponible en https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/String.

3.2.2. Conversiones entre Strings y Numbers

Hacemos uso de métodos tanto de *string* como de *number*. El método `toString` transforma cualquier tipo de dato a *string*. Con el operador de concatenación se puede hacer implícitamente.

```
showMessage (typeof price.toString()); //string
```

Para covertir un *string* a un *number* tenemos los métodos estáticos `parseInt` y `parseFloat` para hacerlo en coma flotante o en entero.

```
showMessage (Number.parseInt ("123")); //123
showMessage (Number.parseFloat ("123.12")); //123.12
```

Ambos métodos parsean el tipo de dato hasta que encuentran un carácter que no sabe procesar, pero no genera un error. En el siguiente ejemplo, el parseo se detiene en el carácter “A”.

```
showMessage (Number.parseFloat ("123A.12")); //123
```

Si al parsear el *string* no podemos obtener un *number* obtenemos un valor `NaN` (Not a Number), que indica que el resultado del parseo no es un *number*.

```
showMessage (Number.parseFloat ("A123.12")); //NaN
```

3.3. Boolean

Las variables *boolean* admiten valores `true` o `false`, pudiendo utilizar el operador unario de negación (!) para cambiar su valor.

```
let changed = true;
showMessage (typeof changed); //boolean
showMessage (changed); //true
showMessage (!changed); //false
```

3.4. Null y undefined

Un valor `undefined` indica que todavía la variable no ha sido asignada a ningún valor en concreto. También podemos asignar el valor `undefined` de forma explícita.

```
changed = undefined;
showMessage (changed); //undefined
console.log (changed); //undefined
```

Un valor `null` indica que la variable ha sido a un valor específico que indica que es nulo, pero la variable si que tiene valor. En el ejemplo, un valor es transformado a *string* como una cadena vacía.

```
changed = null;
showMessage (changed); //
console.log (changed); //null
```

3.5. Formato literal de objetos

Podemos declarar un objeto mediante la notación literal, es decir, de forma textual, utilizando llaves, declarando cada propiedad separada por comas, y la asignación mediante dos puntos. En el ejemplo vemos la declaración de un objeto computadora.

```
let computer1 = {
  brand: 'HP',
  model: 'EliteBook',
  memory: 16,
  SSD: 2,
  discounted: true,
  price: 2000
}
```

La variable de un objeto por si solo solamente apunta a una dirección de memoria. Para acceder a sus propiedades debemos utilizar la notación punto. En el ejemplo vemos como el acceso a la variable nos muestra un texto indicando que se trata de un objeto, y por último podemos acceder al valor de una propiedad determinada.

```
showMessage (computer1); //[object Object]
showMessage (computer1.model); //EliteBook
```

4. Flujo de programa

Creamos un fichero ***04_flow.js*** para realizar los ejemplos y lo importamos en la página.

```
<script src="./js/util.js"></script>
<script src="./js/home.js"></script>
<script src="./js/03_operators.js"></script>
<script src="./js/04_flow.js"></script>
```

4.1. Condicional if-else

Ejecuta código en función de una expresión de entrada en forma de valor booleano mediante operadores de comparación, si es verdadera se ejecuta el bloque de contenido y sino no se ejecuta. El bloque de la clausula `else` se ejecutará en caso de no cumplirse la condición.

```
if (true){
  showMessage (true); // true
} else {
  showMessage (false);
}
```

La siguiente tabla describe los operadores de comparación que podemos utilizar.

Operador	Descripción
Igual (==)	Devuelve true si los operandos son iguales.
No es igual (!=)	Devuelve true si los operandos no son iguales
Estrictamente igual (===)	Devuelve true si los operandos son iguales y del mismo tipo. También se aplica a objetos.
Desigualdad estricta (!==)	Devuelve true si los operandos son del mismo tipo pero no iguales, o son de diferente tipo.
Mayor que (>)	Devuelve true si el operando izquierdo es mayor que el operando derecho.
Mayor o igual que (>=)	Devuelve true si el operando izquierdo es mayor o igual que el operando derecho.
Menor que (<)	Devuelve true si el operando izquierdo es menor que el operando derecho.
Menor o igual (<=)	Devuelve true si el operando izquierdo es menor o igual que el operando derecho.

Tabla 2 Operadores de comparación

Veamos un ejemplo de operadores de igualdad y de igualdad estricta. En el primer ejemplo comparamos un *number 5* con un *string* "5". El intérprete realiza un casting de forma implícita que permite hacer la comparación.

```
if (5 == "5"){
  showMessage(true); //true
} else {
  showMessage(false);
}
```

En el segundo ejemplo no se realiza este casting y por tanto no son iguales.

```
if (5 === "5"){
  showMessage(true);
} else {
  showMessage(false); //false
}
```

Lo ideal es siempre utilizar el operando estricto para tener no solo los valores, sino los tipos de las variables también.

```
if (5 === 5){
  showMessage(true); //true
} else {
  showMessage(false);
}
```

Debemos tener en cuenta en las comparaciones de valores en coma flotante que el intérprete no es preciso en este tipo de comparaciones, porque podríamos obtener resultados que no esperamos.

```
if (1.1 + 1.3 <= 2.4){
  showMessage("Aplicar descuento");
} else {
  showMessage("No aplicar descuento");
}
```

4.2. Verdad y falsedad

El intérprete puede transfor valores no booleanos en booleanos si son utilizados en una expresión de comparación o lógica. La siguiente tabla realiza una comparativa entre valores falsos y verdaderos.

Valores falsos	Valores verdaderos
false	true
0	0.5
Cadena vacía “” o ‘’	“0”
null	Cualquier cosa que no sea falso.
undefined	
NaN	

Tabla 3 Valores falsos y verdaderos

```
if (!0) showMessage("0");
if (!"") showMessage("cadena vacía");
if (!null) showMessage("null");
if (!undefined) showMessage("undefined");
if (!NaN) showMessage("NaN");
```

4.3. Encadenar if…else

Podemos encadenar tantas clausulas `if` con `else` como necesitemos. El siguiente ejemplo muestra cómo sería la clasificación de un producto en función del ratio medio de los usuarios de una web.

```
let rate = 3;
if (rate === 5){
  showMessage("La nota es excelente");
} else if (rate === 4){
  showMessage("La nota es muy buena");
} else if (rate === 3){
  showMessage("La nota buena");
```

```
} else if (rate === 2){
  showMessage("La nota es regular");
} else if (rate === 1){
  showMessage("La nota es muy mala");
} else {
  showMessage("La nota es pésima");
}
```

4.4. Sentencia switch

Podemos simplificar el código del ejemplo del punto anterior con la sentencia `switch`, que permite chequear el valor de una variable múltiples veces hasta que encontrar su valor para realizar una ejecución condicional.

```
rate = 1;
switch (rate) {
  case 5:
    showMessage("La nota es excelente");
    break;
  case 4:
    showMessage("La nota es muy buena");
    break;
  case 3:
    showMessage("La nota buena");
    break;
  case 2:
    showMessage("La nota es regular");
    break;
  case 1:
    showMessage("La nota es muy mala");
    break;
  default:
    showMessage("La nota es pésima");
}
```

La sentencia `break` permite cortar la ejecución de cada uno de los casos, una vez ejecutado su código. Siempre debemos añadir una sentencia `default` para indicar un caso por defecto en la comparación.

4.5. Operadores lógicos

La siguiente tabla muestra los operadores lógicos que podemos utilizar en expresiones condicionales compuestas.

Operador	Uso	Descripción
AND Lógico (&&)	expr1 && expr2	Devuelve true si ambos operandos son true; de lo contrario, devuelve false.
OR lógico (||)	expr1 || expr2	Devuelve true si alguno de los operandos es true; si ambos son falsos, devuelve false.
NOT lógico (!)	!expr	Negamos el valor obtenido en la expresión.

Tabla 4 Operadores lógicos

A continuación, tenemos dos ejemplos de uso en un condicional.

```
rate = 2.5;
if (rate >= 2 && rate <= 4){
  showMessage("La nota está entre 2 y 4");
}

rate = 4.5;
price = 2001;
if (rate < 4 || price > 2000){
  showMessage("Mala relación calidad precio");
}
```

Podemos anidar los operadores lógicos tanto como sea necesario.

```
rate = 4.5;
price = 1900;
if (!(rate < 4 || price > 2000)){
  showMessage("Buena relación calidad precio");
}
```

Hay veces que la expresión de lógica que estamos utilizando puede difícil de entender. Podemos simplificar una negación global por una equivalente negando los operandos individualmente, y alternando *and* por *or* y viceversa. En el siguiente código vemos que un producto con buena relación calidad y precio su ratio no puede ser menor de 4 y su precio no puede ser menor de 2000, siendo equivalente a la del ejemplo anterior.

```
rate = 4.5;
price = 1900;
if (!(rate < 4) && !(price > 2000)){
  showMessage("Buena relación calidad precio");
}
```

La siguiente tabla muestra expresiones equivalentes entre si utilizando operadores lógicos de negación globales.

Expresión con negación global	Expresión equivalente
!(expre1 && expre2)	!expre1 || !expre2
!(expre1 || expre2)	!expre1 && !expre2

Tabla 5 Expresión equivalentes con operadores lógicos globales de negación

4.6. Ejecución de instrucciones con operadores lógicos

Los operadores lógicos en realidad devuelven el valor de uno de los operandos especificados. Podemos utilizar los operadores lógicos para asignar valores no booleanos a variables, mediante la transformación a booleano. Veamos algunos ejemplos.

4.6.1. Ejemplos de and

Devuelve expre1 si se puede convertir a *false*; de lo contrario, devuelve expre2.

```
console.log (3+3 && (12)); // expr1 == t, devuelve expr2, 12
console.log (3-3 && (12)); // expr1 == f, devuelve expr1, 0
console.log ('Perro' && 'Gato');  // expr1 == t, devuelve expr2, 'Gato'
console.log ('' && 'Cat');    // expr1 == f, devuelve expr1, ''
```

Este tipo de expresiones la podemos utilizar en una sentencia para ejecutar código condicionalmente. En función de la expre1, podemos ejecutar expre2. Veamos un ejemplo. Solo se invoca la función en el caso de que expre1 sea *true*.

```
(3 + 3) && showMessage ("Condición true");
(3 - 3) && showMessage ("Condición false");
```

4.6.2. Ejemplo de or

Devuelve expre1 si se puede convertir a *true*; de lo contrario, devuelve expre2.

```
console.log (3+3 || (12)); // expr1 == t, devuelve expr1, 6
console.log (3-3 || (12)); // expr1 == f, devuelve expr2, 12
console.log ('Perro' || 'Gato');  // expr1 == t, devuelve expr1, 'Perro'
console.log ('' || 'Cat');    // expr1 == f, devuelve expr1, 'Gato'
```

Como con el operador *and* podemos ejecutar código condicionalmente con *or*. En este caso la función se invoca si la expre1 es *false*.

```
(3 - 3) || showMessage ("Condición false");
(3 + 3) || showMessage ("Condición true");
```

4.7. Operador ternario

Este operador utiliza tres operandos. El primero es una expresión lógica o de comparación. En función de su resultado se ejecuta el código del segundo operador, y si es falto se ejecuta el del tercero. Este es otro ejemplo de uso de condicionales sin sentencia `if`.

```
price = 15;
(price >=10) ? showMessage ("Por encima") : showMessage ("Por debajo");
```

También se puede utilizar en asignaciones.

```
message = (price >=10) ? "Por encima" : "Por debajo";
showMessage (message);
```

4.8. Sentencia let en el ámbito de un bloque

El siguiente código muestra un condicional y en su interior definimos una constante o una variable mediante `let`, esta variable se puede acceder desde dentro del bloque sin problema.

```
if (true) {
  let value1 = "Valor interno 1";
  showMessage (value1);
}
```

Si quisiéramos acceder desde fuera del bloque del condicional a la variable, el intérprete genera un error indicando que la variable no está definida.

```
console.log(value1);
```

Si utilizasemos la sentencia `var`, el ámbito de esta variable se mantiene fuera del bloque.

```
if (true) {
  var value2 = "Valor interno 2";
  showMessage (value2);
}
console.log(value2); // Varlo interno 2
```

4.9. Bucle for

Permite repetir el bloque del bucle un número de veces. La declaración consta de tres cláusulas separadas por punto y coma:

1. Definición de variables.
2. Condición de salida.
3. Incremento de variables.

El siguiente ejemplo muestra un bucle que incrementa la variable *i* de uno en uno mientas su valor es menor de 5.

```
for (let i = 0; i < 5; i++){
  console.log ("i: " + i); //0, 1, 2, 3, 4
}
```

Lo podemos rehacer decrementando la variable.

```
for (let i = 5; i > 0; i--){
  console.log ("i: " + i); //5, 4, 3, 2, 1
}
```

También podemos declarar más de una variable, utilizarlas en la condición de salida e incrementarlas.

```
for (let i = 0, j = 0; i < 5 || j < 15; i++, j = j + 2 ){
  console.log ("i: " + i); //0, 1, 2, 3, 4, 5, 6, 7
  console.log ("j: " + j); //0, 2, 4, 6, 8, 10, 12, 14
}
```

4.10. Bucle while

Es utilizado para chequear una condición de comparación o lógica, pudiendo ser tan compleja como deseemos. El caso típico de uso es cuando no conocemos a priori, el número de veces que se tiene que ejecutar el bloque del bucle. El siguiente código muestra la ejecución de un bucle `for` utilizando un bucle `while`.

```
let i = 4;
while (i > 0){
  console.log ("i: " + i); //4, 3, 2, 1
  --i;
}
```

Debemos darnos cuenta, que en este caso concreto la variable de control del bucle quedaría disponible fuera del bloque del bucle.

El número de veces que se puede iterar sobre el bloque de un bucle `while` es 0 o más veces.

4.11. Bucle do … while

En este bucle la condición se chequea al final del bloque en lugar de al comienzo, por lo tanto, el número de veces que se ejecuta el bloque es 1 o más veces. Por lo demás, el bucle se comporta exactamente igual que un bucle `while`.

```
let j = 4;
do {
```

```
  console.log ("j: " + j); //4, 3, 2, 1
  --j;
} while (j > 0)
```

5. Funciones

Para implementar los ejercicios creamos el fichero ***05_function.js*** y lo importamos en la página.

```
<script src="./js/util.js"></script>
<script src="./js/home.js"></script>
<script src="./js/03_operators.js"></script>
<script src="./js/04_flow.js"></script>
<script src="./js/05_function.js"></script>
```

5.1. Asignación de funciones a variables

Las funciones en JavaScript son objetos y, por tanto, pueden ser asignados a una variable. La variable actúa como nombre de la función para que pueda ser invocada.

```
let myFunction = function (){
  showMessage ("Invocación de la función");
}
myFunction();
```

Un objeto de tipo función devuelve como tipo *function*.

```
showMessage (typeof myFunction);
```

5.2. Valores de retorno

Las funciones pueden devolver un valor utilizando la sentencia `return`, siendo su tipo irrelevante.

```
function getCode(value){
  let code = value * 12;
  return code;
}
showMessage (getCode(11)); //132
```

5.3. Ámbito de funciones

El ámbito de las variables definidas dentro de la función queda restringido a la función. No podemos acceder desde fuera de una función al contenido de las variables definidas en ella. El siguiente código generará un error al ser ejecutado por el intérprete.

```
showMessage (code);
```

Sin embargo, desde dentro de una función podemos acceder a variables definidas en ámbitos externos a la función. En el siguiente ejemplo tenemos una variable externa a nivel global que podremos utilizar en el interior de nuestra función.

```
let param = 12;
function getCodeWithParam(value){
  let code = value * param;
  return code;
}
```

```
showMessage (getCodeWithParam(12)); //144
```

5.4. Funciones internas

Dentro de una función podemos definir funciones internas, las cuales funcionan como la declaración de una variable. Su ámbito está limitado a la función contenedora, por lo que no está accesible desde el exterior.

Como podemos ver en este ejemplo, tenemos dos variables declaradas con el mismo nombre, *param2*, una interna y otra global. Al referenciar una variable el intérprete busca primero la declaración en el ámbito local, y en caso de no encontrarla lo hace en el externo.

```
let param2 = 48;
function getCodeInternalFunction(value){

  function getKey(){
    let param2 = 12;
    console.log ("Param2 en f unción interna: " + param2); //12
    return param2;
  }

  let code = value * getKey();
  console.log ("Param2 en f unción interna: " + param2); //48
  return code;
}
showMessage (getCodeInternalFunction(13)); //156
```

Si en el código anterior eliminaremos la declaración de la variable *param2* en la función interna, el código seguiría funcionando sin error, ya que la función `getKey` buscaría la declaración en el ámbito superior, en este caso, lo encontraría en la función `getCodeInternalFuction`.

6. Arrays

Para trabajar los arrays vamos a crear un fichero ***06.array.js*** y lo tendremos que importar en la página HTML.

```
<script src="./js/util.js"></script>
<script src="./js/home.js"></script>
<script src="./js/03_operators.js"></script>
<script src="./js/04_flow.js"></script>
<script src="./js/05_function.js"></script>
<script src="./js/06_array.js"></script>
```

6.1. Declaración de arrays

Para declarar un array utilizamos los corchetes, separando los elementos mediante comas. Un array es fácilmente transformable a *string*, ya sea de forma automática como en el siguiente ejemplo, o utilizando el método `toString`.

```
let letters = ['a', 'b', 'c', 'd'];
showMessage(letters); // a,b,c,d
```

A diferencia de otros lenguajes, JavaScript permite el uso de arrays con tipos de datos heterogéneos.

```
let lettersAndNumbers = ['a', '1', 'c', '2'];
showMessage(lettersAndNumbers); // a,1,c,2
```

Un array internamente es un tipo de objeto. JavaScript no dispone del tipo array.

```
showMessage(typeof lettersAndNumbers); // object
```

Por lo tanto, si queremos saber si una variable es un array debemos utilizar el método estático `Array.isArray()` que devuelve un booleano.

```
showMessage(Array.isArray (lettersAndNumbers)); // true
```

6.2. Acceso a ítems

Para acceder a un elemento concreto del array, tenemos que indicar su posición entre corchetes, teniendo en cuenta que la primera posición es enumerada con el 0.

```
showMessage(lettersAndNumbers[2]); // c
```

Si accedemos a un elemento del array que no está definido obtenemos *undefined*.

```
showMessage(lettersAndNumbers[20]); // undefined
```

Si queremos añadir un nuevo elemento al array lo podemos hacer utilizando la propiedad `length`.

```
lettersAndNumbers[lettersAndNumbers.length] = 'e';
showMessage(lettersAndNumbers); // a,1,c,2,e
```

Por último, si asignamos un elemento más allá del tamaño de un array, el resto de posiciones se autoasignan con valore *undefined*.

```
lettersAndNumbers[7] = 'z';
showMessage(lettersAndNumbers); // a,1,c,2,e,,,z
```

6.3. Métodos de manipulación de array

6.3.1. Trabajo en la cola del array

El método `push` permite añadir elementos al final del array. El método permite añadir un número arbitrario de elementos.

```
letters.push('e', 'f', 'g', 'h');
showMessage(letters); // a,b,c,d,e,f,g,h
```

Para obtener el último elemento de la cola del array utilizamos el método `pop` y lo asignamos a una variable para utilizarlo posteriormente, aunque este paso no es obligatorio.

```
let lastLetter = letters.pop();
showMessage(letters); // a,b,c,d,e,f,g
```

6.3.2. Trabajo con el inicio del array

Los métodos que trabajan con el inicio del array funcionan similarmente a los anteriores. Para conseguir el primer valor del array tenemos el método `shift`.

```
let firstLetter = letters.shift();
showMessage(letters); // b,c,d,e,f,g
```

Para añadir al inicio del array tenemos el método `unshift`.

```
letters.unshift('a', 'a');
showMessage(letters); // a,a,b,c,d,e,f,g
```

6.3.3. Trabajo utilizando posiciones

El método `slice` permite obtener un nuevo array con el subconjunto de elementos del array indicando sus posiciones.

```
let newLetters = letters.slice(1,4);
showMessage(newLetters); // a,b,c
```

El método `splice` es más complicado, permite añadir y eliminar elementos simultáneamente en un array. El método utiliza tres parámetros:

1. La posición sobre la que se ejecutarán los cambios.
2. El número de elementos que se van a eliminar.
3. El resto de los parámetros son los datos que se añadirán al array.

El siguiente ejemplo añade en el array a partir de la posición 1, tres elementos sin eliminar ningún elemento previo.

```
newLetters.splice (1, 0, '1', '2', '3');
showMessage(newLetters); // a,1,2,3,b,c
```

El siguiente ejemplo elimina el elemento de la posición 2, y añade un nuevo elemento en dicha posición.

```
newLetters.splice (2, 1, '4');
showMessage(newLetters); // a,1,4,3,b,c
```

Por último, podemos eliminar varios elementos del array para dejarlo como al comienzo.

```
newLetters.splice (1, 3);
showMessage(newLetters); // a,b,c
```

6.4. Buscar elementos en un array

Además de utilizar un bucle para recorrer una array, JavaScript tiene otros métodos para obtener el elemento que estamos buscando dentro del array.

6.4.1. Obtener posición de un elemento

Para tipos primitivos, *string*, *number* y *boolean*, podemos utilizar el método `indexOf` para dado un dato, obtener su posición. Es importante entender que para objetos este método no funcionaría como esperamos, ya que la comparación se haría a través de la dirección a la que apunta el objeto.

En este ejemplo obtenemos la posición de un dato dentro de un array.

```
showMessage(newLetters.indexOf('c')); // 2
```

Si el dato no se encuentra, el método devuelve -1.

```
showMessage(newLetters.indexOf('d')); // -1
```

El método `indexOf` realiza la búsqueda desde el inicio del array. Si queremos buscar empezando por el final tenemos el método `lastIndexOf`.

```
showMessage(letters.indexOf('a')); // 0
```

```
showMessage(letters.lastIndexOf('a')); // 1
```

6.4.2. Recorrer un array

El método `forEach` permite recorrer todos los elementos de un array. Como parámetros utiliza una función que será invocada por cada elemento del array. Tenemos que recordad que una función es un objeto, y por tanto podemos pasarlo como parámetro a un método o una función.

Esta función, aunque puede recibir más parámetros, como mínimo debe declarar uno que obtendrá como argumento el elemento con el que ha sido invocada. En el ejemplo este parámetro es `item`.

```
let numbers = [1, 2, 32, 14, 3, 23, 5];
numbers.forEach (function (item){
  console.log("Elemento: " + item);
});
```

6.4.3. Filtrar elementos

El método `filter` funciona igual que `forEach`, pero la función que recibe como parámetro debe devolver un boolean indicando si se selecciona el elemento o no, implementando el mecanismo de selección de elementos. El método `filter` devuelve un array con los elementos seleccionados.

```
let greaterNumbers = numbers.filter(function (item){
  return item > 10;
});
showMessage(greaterNumbers); // 32,14,23
```

7. Objetos

Creamos el fichero ***07_object.js*** para contener los ejemplos y lo importamos en la página.

```
<script src="./js/util.js"></script>
<script src="./js/home.js"></script>
<script src="./js/03_operators.js"></script>
<script src="./js/04_flow.js"></script>
<script src="./js/05_function.js"></script>
<script src="./js/06_array.js"></script>
<script src="./js/07_object.js"></script>
```

7.1. Propiedades

Ya hemos visto la sintaxis de creación de nuestros propios objetos con un formato literal que consiste en definir las propiedades del objeto con sus valores. Aunque todavía no lo hemos trabajado, las propiedades podrían ser nuevos objetos o un array.

El siguiente ejemplo muestra la declaración de un objeto que representa un ordenador. Su propiedad `hardDisks` es un objeto que cuenta con dos arrays como propiedades.

```
let computer2 = {
  brand: 'HP',
  model: 'EliteBook',
```

```
  memory: 16,
  hardDisks: {
    SSD: [1024],
    HDD: [2048,1024]
  },
  discounted: true,
  price: 2000
}
```

Para acceder o modificar las propiedades utilizamos la **notación punto**.

```
showMessage (computer2.model); //EliteBook
```

En el caso de que la propiedad sea otro objeto o un array seguimos utilizando la notación punto y el acceso a través del índice del elemento concreto.

```
showMessage (computer2.hardDisks.HDD); //2048,1024
showMessage (computer2.hardDisks.HDD[1]); //1024
```

También podemos utilizar el acceso a una propiedad mediante el *string* con el nombre de la propiedad.

```
showMessage (computer2['model']); //EliteBook
```

7.2. Métodos

Un método es una propiedad que tiene asociada una función. Desde los métodos podemos acceder o modificar el resto de las propiedades del objeto mediante la palabra reservada `this`, que representa el propio objeto.

```
let computer3 = {
  brand: 'HP',
  model: 'EliteBook',
  memory: 16,
  hardDisks: {
    SSD: [1024],
    HDD: [2048,1024]
  },
  discounted: true,
  price: 2000,
  showInfo: function(){
    return this.brand + " " + this.model + " (" + this.memory + "GB)";
  }
}
showMessage (computer3.showInfo()); //HP EliteBook (16GB)
```

7.3. Pasando objetos por referencia a funciones

Una variable con un dato primitivo cuando se pasa como argumento a una función es copiada por el intérprete y asignada al parámetro que define la función, es decir, aunque modifiquemos su contenido internamente en la función, su valor no cambia externamente porque se trata de una copia. Este tipo de paso de datos es conocido como paso por valor.

```
function changeValue (value){
  console.log("Valor antes del cambio: " + value); //abcde
```

```
  value = 12345;
  console.log("Valor después del cambio: " + value); //12345
}
let value = "abcde";
changeValue (value);
console.log("Valor fuera de la función: " + value); //abcde
```

Los objetos son pasados por referencia a una función, es decir, el parámetro recibe como argumento la dirección de memoria donde se encuentra almacenado el objeto, por tanto, cada cambio que hagamos en el objeto se hará directamente en dicha dirección, quedando el cambio permanente en el dato.

La siguiente función recibe un objeto con la información de un ordenador, y le cambia el nombre y la marca. Al acceder a la información del objeto fuera de la función, las propiedades modificada prevalecen ya que estos cambios se han hecho directamente en la memoria donde se almacena el objeto.

```
function changeComputer(computer){
  computer.brand = "Dell";
  computer.model = "XPS";
  computer.memory = 32;
}
changeComputer(computer3);
showMessage (computer3.showInfo()); //Dell XPS (32GB)
```

7.4. Styling DOM

Cada objeto DOM tiene una propiedad `style` con las propiedades CSS que están asociadas con el objeto. Esta propiedad es realmente otro objeto donde asignaremos las propiedades CSS. El nombre de las propiedades coincide con la especificación CSS, salvo que las propiedades CSS que utilizan guiones, se traducen por su formato *camelCase*. Por ejemplo la propiedad *font-weight* se traduce por `fontWeight` en JavaScript.

En el fichero *util.js* añadimos una función que recibe un identificador de un elemento, para añadirle un borde rojo a ese elemento.

```
function setRedBorderInElement (id){
  document.getElementById(id).style.border = "3px solid red";
}
```

La invocamos para chequear su funcionamiento.

```
setRedBorderInElement('message');
```

El objeto `classList` de un objeto DOM contiene la lista de clases CSS aplicadas al objeto DOM. Su método contains nos indica si una clase determinada forma parte de la lista, y los métodos `remove` y `add` permite eliminar y añadir clases a la lista respectivamente.

7.5. Detectando eventos

Los elementos HTML que tenemos en la página están sujetos a la interacción con el usuario, esta interacción se produce a través de eventos. Algunos ejemplos de eventos es hacer clic en algún elemento de la página, pulsar una tecla, cargar una página o pasar el ratón por encima de un elemento entre muchos otros. Vamos a ver como trabajar capturar estos eventos desde JavaScript.

En la página HTML, buscamos el enlace con texto "ver artículo", que tiene forma de botón. Le añadimos un identificador que nos permita acceder al objeto DOM.

```
<a id="button" class="btn btn-default" href="#">Ver artículo</a>
```

Desde nuestro fichero de pruebas *07_object.js*, pasamos a definir la variable con el objeto DOM.

```
let button = document.getElementById("button");
```

Un *listener* es una función que será ejecutada en respuesta a un evento generado en la página. El método `addEventListener` permite añadir un *listener* asociado a un tipo de evento concreto a un objeto DOM. Asociamos al evento "click" del objeto anterior una función.

La función añadirá un borde al elemento *message* de nuestra página.

```
let button = document.getElementById("button");
button.addEventListener ("click", function (){
  setRedBorderInElement('message');
});
```

Modificamos el *listener* anterior para cambiar la funcionalidad. El framework **Bootstrap,** a través de la clase *d-none*, asigna la propiedad *display* con valor *none* para eliminar el elemento del árbol de renderización.

El objeto `classList` de un elemento representa el conjunto de clases aplicadas al elemento. Tiene tres métodos que podemos utilizar:

- `contains`: Lo utilizamos para saber si una clase está añadida en el elemento.
- `remove`: Elimina una clase del elemento.
- `add`: Añade una clase del elemento.

Por último, mediante la propiedad `textContent` podemos modificar el texto de un elemento.

En el *listener* vamos a chequear si elemento *categories* tiene aplicada la clase *d-none*. Si la tiene la eliminamos y si no la tiene la añadimos. Cambiaremos el texto del botón para que el usuario sea consciente de cuál será la acción que se producirá al cliquear en el botón.

```
let button = document.getElementById("button");
button.addEventListener ("click", function (){
  let categories = document.getElementById("categories");

  if (categories.classList.contains("d-none")){
    categories.classList.remove("d-none");
    button.textContent = "Ocultar categorías";
  } else {
    categories.classList.add("d-none");
    button.textContent = "Mostrar categorías";
  }
});
```

8. Ámbitos

Creamos e importamos el fichero ***08_scope.js***.

```
<script src="./js/util.js"></script>
<script src="./js/home.js"></script>
<script src="./js/03_operators.js"></script>
<script src="./js/04_flow.js"></script>
<script src="./js/05_function.js"></script>
<script src="./js/06_array.js"></script>
<script src="./js/07_object.js"></script>
<script src="./js/08_scope.js"></script>
```

8.1. Ámbito global

El ámbito global de la aplicación incluye cualquier declaración de variables o funciones realizadas en ficheros JavaScript, no incluidos dentro de otra función. Todas estas declaraciones estarán accesibles desde el interior de cualquier función ejecutada en la página.

```
let computerId = 9876;
function showComputerId(){
  showMessage(computerId);
}
showComputerId();
```

Una buena práctica a la hora de gestionar tus variables globales es agruparlas en una única declaración. El siguiente código muestra cómo podríamos utilizar un objeto declarado en una constante a nivel global, que agrupe todas las variables globales que estemos utilizando.

```
const APP = {
  computerId: 9876,
  userName: "pablo",
  profile: "admin"
}

function showGlobalInfo(){
  showMessage (APP.computerId + " " + APP.userName); //9876 pablo
}
showGlobalInfo();
```

8.2. Modo estricto

El modo estricto permite corregir ciertas licencias que el lenguaje JavaScript permite, pero que pueden producir código menos seguro y provocar errores difíciles de localizar. Una de las opciones que nos obliga el uso del modo estricto es que no podemos utilizar una variable sin antes declararla. Veamos un ejemplo.

El siguiente código muestra la asignación de una variable sin antes haberla declarado. Lo que ocurre realmente es que la variable es asignada como una propiedad al objeto window, el cual representa la ventana del navegador que estamos utilizando, y por tanto es accesible desde cualquier ámbito de la página o la aplicación.

```
function strictModeExample(){
```

```
  categoryId = 123456;
  showMessage(window.categoryId);
}
strictModeExample();
```

Para solventar este tipo de problemas, utilizamos el modo estricto a través de una directiva al inicio de una función, para limitar el modo estricto dentro de la función, o en la primera línea del archivo, para que se aplique a todo su contenido.

```
function strictModeExample(){
  "use strict";
  categoryId = 123456;
  showMessage(window.categoryId);
}
strictModeExample();
```