

Enlaces web de interés

-  **IBM Developers** - <https://developer.ibm.com/technologies/web-development/>
(tecnologías open source para construir aplicaciones web)
-  **Documentación Microsoft** - <https://docs.microsoft.com/es-es/windows/web/>
(desarrollo web en Windows)
-  **Google Scholar** - <https://scholar.google.es/>
(buscador de Google especializado en documentos académicos)
-  **Comunidad JavaScript** - <https://www.javascript.com/>
(recursos para principiantes)
-  **Technopedia** - <https://www.techopedia.com/>
(repositorio de conceptos sobre tecnología)
-  **Documentación Oracle** - <https://docs.oracle.com/>
(guía para desarrolladores del lado cliente)



Introducción a la programación en JavaScript

Objetivos

- Conocer los detalles de la identidad de JavaScript que lo convierten en un referente y obtener una foto global del lenguaje.
- Identificar dónde y cómo se puede ejecutar el código JavaScript.
- Estudiar la estructura principal de los programas.
- Saber crear variables de los tipos correctos y operar con ellas entendiendo las conversiones entre tipos.
- Descubrir la variedad de mensajes disponibles y saber aplicarlos correctamente en cada situación.
- Aplicar todos los operadores de forma correcta conociendo sus matices.
- Entender cómo definir el flujo de ejecución de un programa utilizando con solvencia las estructuras de control del lenguaje.
- Ser capaz de realizar un seguimiento de la ejecución de los programas.

Contenidos

- 2.1. Conceptos básicos
- 2.2. Variables
- 2.3. Entrada y salida en el navegador
- 2.4. Operadores
- 2.5. Estructuras de control

Introducción

Como en cualquier lenguaje de programación, los conceptos básicos sobre los que se sostiene toda su arquitectura son fundamentales para entender el resto de los desarrollos. El caso de JavaScript no es una excepción, ya que existen algunos matices pequeños pero importantes que se verán en esta sección y que serán determinantes para escribir un código correcto y eficiente.

Se realizará un primer recorrido por el lenguaje, partiendo de conceptos básicos teóricos, hasta llegar a estructuras repetitivas complejas, pasando por el estudio de las variables, los tipos de datos, los operadores y los mensajes.

2.1. Conceptos básicos

Ha llegado el momento de entrar en harina. Lo primero será empezar a escribir código JavaScript y repasar los fundamentos básicos del lenguaje a través de sus instrucciones más importantes. Esta unidad es completamente crucial para poder abordar con éxito el desarrollo de aplicaciones web en entorno cliente con JavaScript.

Se empezará por conocer los rasgos generales que definen el lenguaje y a partir de ahí se irán desgranando todas sus características fundamentales.

2.1.1. El ADN de JavaScript

JavaScript es un lenguaje de programación ligero, interpretado, basado en prototipos, case sensitive, débilmente tipado, multiparadigma, monohilo, dinámico y con soporte para la programación orientada a objetos, imperativa y declarativa. Casi nada.

Seguramente haya uno o varios conceptos de la frase anterior que no se dominan. No hay motivo para preocuparse; al finalizar este libro se tendrá un conocimiento profundo de todos ellos. De momento, se van a explicar brevemente cada uno de ellos para tener una foto global del lenguaje que se está aprendiendo:

- **Ligero:** está diseñado para ocupar poco espacio en memoria, es fácil de implementar y cuenta con una sintaxis y una semántica simples, por lo que se puede aprender en poco tiempo.
- **Interpretado:** significa que se lee y se ejecuta cada línea de código de forma secuencial, al contrario que los lenguajes compilados cuyas líneas de código son convertidas en su conjunto a lenguaje máquina para ser ejecutados posteriormente.
- **Basado en prototipos:** se trata de una variante de la programación orientada a objetos en la que los objetos no se crean instanciando clases sino clonando otros objetos o creándolos directamente. Lo que debe recordarse de esta característica es que en JavaScript todo es un objeto.

- **Case sensitive:** es sensible a mayúsculas y minúsculas, por lo que rojo, Rojo y ROJO serán tres identificadores distintos.
- **Débilmente tipado:** el lenguaje no necesita conocer al detalle con qué tipo de dato se está trabajando en cada momento. Es lo suficientemente inteligente para deducirlo o realizar conversiones entre tipos de datos a lo largo de la ejecución del programa.
- **Multiparadigma:** a diferencia de otros lenguajes, permite programar aplicaciones completas desde cero aplicando muy distintos paradigmas de programación.
- **Monohilo:** un único hilo de ejecución se encarga de realizar el trabajo de interpretación del código, de forma que se dificultan los interbloqueos y se favorece el uso de las llamadas asíncronas.
- **Dinámico:** el lenguaje es capaz de cambiar importantes características del programa, como la estructura de un objeto o el tipo de una variable, mientras se está ejecutando.
- **Programación orientada a objetos:** un modelo de programación mucho más cercano al mundo real donde se modelan patrones a través de clases y se instancian en forma de objetos. Los objetos se relacionan entre sí para alcanzar los objetivos de las aplicaciones.
- **Programación imperativa:** un modelo de programación consistente en una secuencia de instrucciones que determinan qué debe hacer la máquina en cada momento paso a paso. Se centran en el «cómo» de la solución.
- **Programación declarativa:** un modelo de programación que consiste en describir qué se quiere obtener al finalizar la ejecución del programa, en lugar de cómo se quiere obtener. Se centran en el «qué» de la solución.

2.1.2. Dónde ejecutar código JavaScript

JavaScript es un lenguaje de programación que ha evolucionado mucho en los últimos años. Una de las mayores innovaciones que ha experimentado ha sido precisamente dónde y cómo se ejecuta.

Atendiendo a la arquitectura de la aplicación, se tiene lo siguiente:

- **JavaScript del lado cliente:** proporciona objetos para gestionar el navegador y el modelo de objetos del documento (DOM, Document Object Model) del mismo. El DOM es la estructura interna que entiende JavaScript cuando lee un documento HTML. De esta forma se puede controlar y responder a los eventos del usuario (clic de ratón, pulsación de una tecla...), formularios, navegación entre páginas o cualquier otro elemento de la interfaz de usuario.
- **JavaScript del lado servidor:** amplía el núcleo del lenguaje extendiéndolo con objetos de interés para poder ser ejecutado en un servidor, normalmente a través de Node.js. Así, es posible comunicarse con una base de datos, gestionar ficheros del servidor u ofrecer respuestas a solicitudes de aplicaciones.

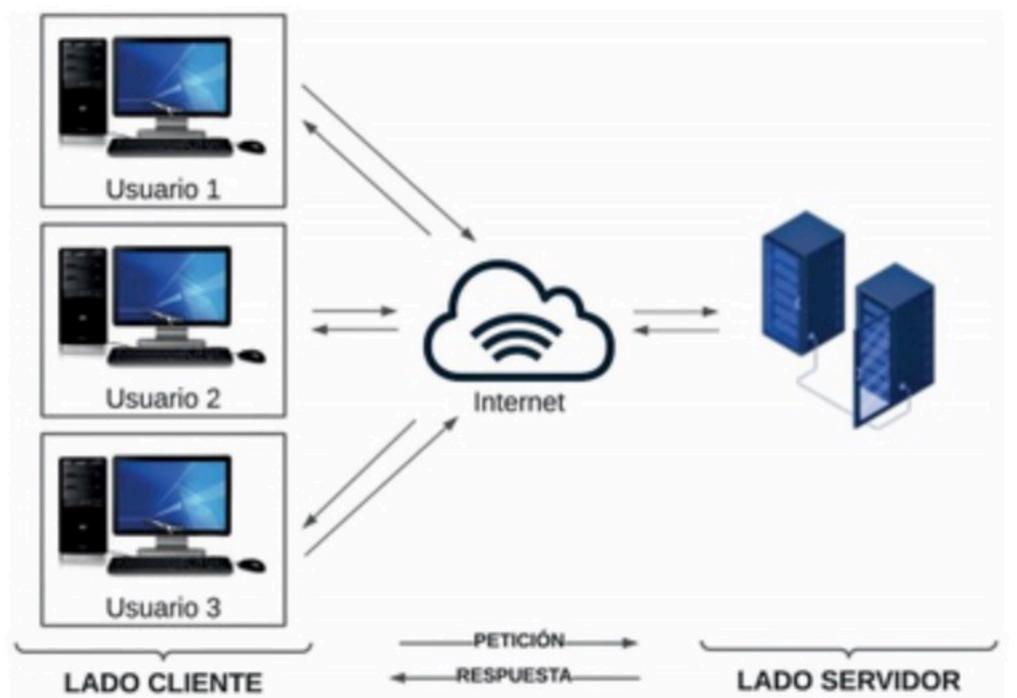


Figura 2.1. Esquema simplificado de una arquitectura cliente-servidor.

Teniendo en cuenta que esta obra trata en exclusiva del desarrollo de aplicaciones web en el lado cliente, quedan fuera del objeto de estudio las capacidades de JavaScript del lado servidor.

Se estudiará, por tanto, dónde ejecutar código JavaScript del lado cliente. Salvo situaciones muy excepcionales, siempre que se ejecute código JavaScript del lado cliente, se utilizará un navegador.

Para ello, puede escribirse el código dentro de alguno de los ficheros que es capaz de leer el navegador, o bien utilizar la consola que incorpora. Aunque, como se irá viendo a lo largo del libro, lo más habitual es combinar las dos opciones en función de las necesidades en cada momento.

De ahora en adelante, y para facilitar la comprensión del código, se utilizará un fichero interpretado por el navegador. Además, siempre que sea de utilidad, se acudirá a la consola para observar algunas salidas que mejorarán la comprensión de los conceptos.

2.1.3. Cómo ejecutar código JavaScript

Como ya se adelantó en el apartado anterior, se utilizará un fichero que sea capaz de interpretar el navegador. Existen muchos tipos de ficheros que podrían servir, aunque lo correcto atendiendo a las normas de estilo es que el código JavaScript se incluya como parte de un fichero HTML o separado en un fichero .js

El lenguaje de marcado HTML ya incluye una etiqueta que permite incorporar el código JavaScript. Se trata de la etiqueta **script**, de la que a continuación se muestra un ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title></title>
</head>
<body>
<script>
document.write("Código JS dentro de HTML");
</script>
</body>
</html>
```

Al ejecutar el código anterior en el navegador se verá un documento en blanco con un único mensaje escrito por el método **document.write**.

Código JS dentro de HTML

Figura 2.2. Resultado que muestra el navegador tras ejecutar el código anterior.

Como puede verse, se ha colocado el código JavaScript dentro de la sección **body** de la web, pero no tiene por qué ser así. Técnicamente podría colocarse en cualquier parte del documento, aunque las alternativas habituales son colocarlo en la sección **head** o incluso como parte del valor de un atributo en etiquetas HTML. Es una norma de estilo colocarlo en el **body** si se va a interactuar con los elementos del DOM; que se coloque en **head** si se van a incluir otras librerías o a declarar estructuras de datos globales; y que se coloque dentro de una etiqueta HTML si se quiere modificar el comportamiento de ese único elemento. Así se haría en el último caso:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title></title>
</head>
<body>
<p onclick="alert('Un mensaje lanzado por JavaScript')">
Esto es un párrafo con un evento asociado.
</p>
</body>
</html>
```

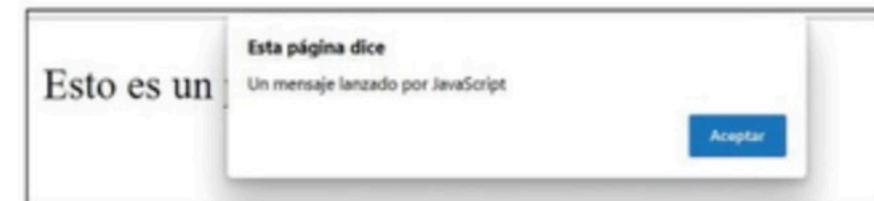


Figura 2.3. Resultado que muestra el navegador tras clicar sobre el texto del párrafo.

Actividad propuesta 2.1

JavaScript incrustado en el navegador

Comprueba que puedes situar el código JavaScript en cualquier parte de tu fichero HTML. Para ello:

- Crea un fichero HTML con varias etiquetas y contenido de prueba.
- Prueba a situar código JavaScript dentro de cada etiqueta y comprueba el resultado en el navegador.

La otra opción que se ha comentado es la más común y la que proporciona mayor modularidad y facilidad de mantenimiento. Se trata de la inclusión de ficheros .js.

La idea es escribir ficheros con únicamente código JavaScript e incluirlos en el código HTML que necesite usarlos. A continuación, un ejemplo.

Primero se crea un fichero nuevo con el nombre deseado, en este caso **micodeigo.js** y se incluye todo el código JavaScript de la aplicación:

```
document.write("Código JS en un fichero externo");
```

A continuación se utiliza la etiqueta **script** para incluir el fichero JavaScript dentro del fichero HTML, en este caso **index.html**, con el que se desea ejecutar el código:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <script src="micodeigo.js"></script>
  <title>Incluyendo JS externo</title>
</head>
<body>
  <p>
    Un párrafo cualquiera.
  </p>
</body>
</html>
```

Así, cuando el navegador ejecute el fichero **index.html** y llegue a la etiqueta **script** buscará el fichero .js en la ruta indicada en el atributo **src**, ejecutará el código que encuentre y continuará interpretando el resto del código HTML.

Código JS en un fichero externo
Un párrafo cualquiera.

Figura 2.4. Resultado que muestra el navegador tras ejecutar el código HTML anterior.

La etiqueta **script** cuenta con algunos otros atributos que pueden aportar una importante funcionalidad extra en determinadas ocasiones, aunque los dos más utilizados son los que se detallan en la Tabla 2.1.

Tabla 2.1. Descripción general de los dos atributos más utilizados por la etiqueta script

Atributo	Descripción
src	Especifica la URL del fichero que se desea cargar.
type	Indica el tipo de fichero que se está utilizando. Si se omite, se entiende que se trata de código JavaScript

Actividad propuesta 2.2

JavaScript desde un fichero externo

Asegúrate de que eres capaz de incluir un fichero JavaScript externo en tu aplicación.

- Crea un fichero HTML con varias etiquetas y contenido de prueba.
- Crea otros ficheros JavaScript donde incluyas algunas instrucciones (distintas en cada fichero) y guárdalos en diferentes carpetas de tu equipo.
- Por último, modifica tu fichero HTML para incluir un fichero JavaScript cada vez y comprueba en el navegador que ejecuta las instrucciones que contiene.

Para finalizar, es importante saber que si el código JavaScript contiene errores o advertencias se puede acceder a ellos a través de la consola del navegador.

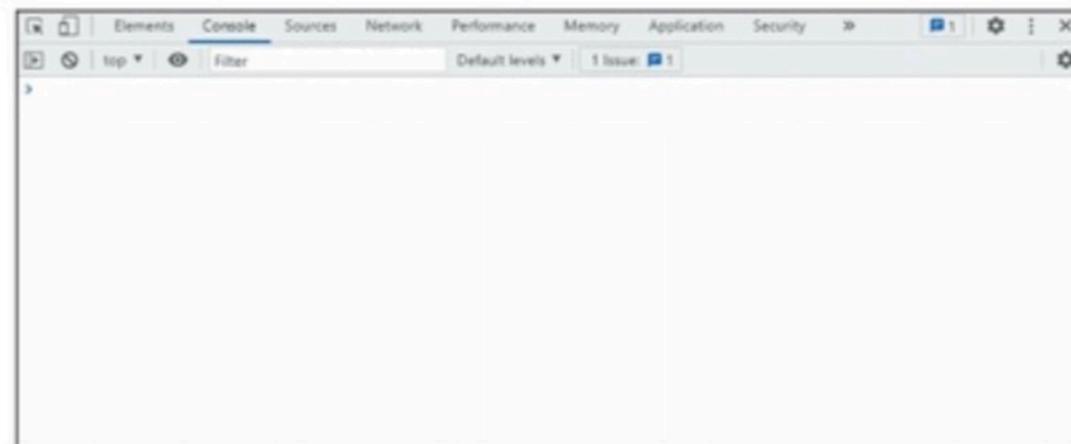


Figura 2.5. Aspecto de la consola en Google Chrome.

Además, puede utilizarse la consola para mostrar mensajes propios que ayuden a realizar un seguimiento de la ejecución del programa mientras se está desarrollando. Es una utilidad que se utilizará mucho para explicar los conceptos básicos del lenguaje, así que es importante familiarizarse con su uso.

Las consolas de todos los navegadores son muy parecidas, aunque difieren en su forma de acceder a ellas.

Tabla 2.2. Atajo de teclado para abrir la consola en los principales navegadores de escritorio

Navegador	Atajo de teclado
Google Chrome	[Ctrl]+[Mayúsculas]+[J] [Comando]+[Opción]+[J]
Microsoft Internet Explorer / Edge	[F12]+Seleccionar Consola
Mozilla Firefox	[Ctrl]+[Mayúsculas]+[K] [Comando]+[Opción]+[K]
Apple Safari	[Comando]+[Opción]+[C]

Una vez abierta se ve un *prompt* que está esperando recibir órdenes. Tan solo hay que escribir o pegar código JavaScript y presionar [Intro]. El navegador ejecutará el código y mostrará el resultado de la ejecución.

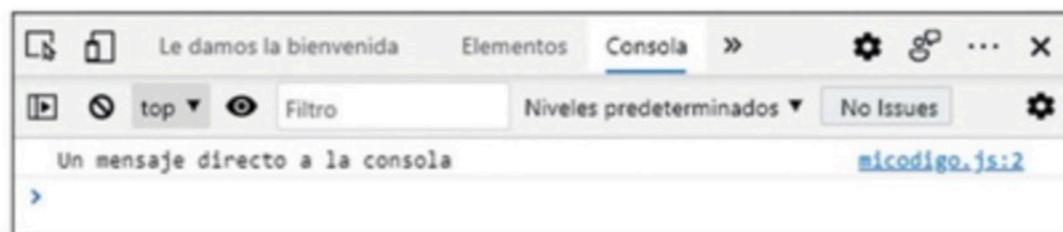
Nota técnica

Se denomina **prompt** al identificador, carácter o símbolo que se muestra habitualmente en las líneas de comandos para hacer saber que está a la espera de recibir órdenes. En el caso de la consola del navegador se muestra el símbolo >.

Si en el ejemplo anterior se añade la siguiente línea de código al fichero **micodigo.js**:

```
document.write("Código JS en un fichero externo");
console.log("Un mensaje directo a la consola");
```

No se verá ningún cambio en la web tras ejecutar **index.html** en el navegador. Sin embargo, al abrir la consola se verá lo que se muestra en la Figura 2.6.

**Figura 2.6.** Nuevo mensaje en la consola tras ejecutar el fichero.

Se puede ver el mensaje que quería lanzarse por medio de la instrucción **console.log**, además del fichero y el número de línea donde se encontraba la instrucción que lo generó. Son detalles que ofrece la consola y que, como se verá a lo largo de este libro, serán de enorme utilidad desarrollando aplicaciones JavaScript.

Actividad propuesta 2.3

Consola del navegador

Vamos a descubrir el funcionamiento de la consola de tu navegador:

- Abre la consola de tu navegador y familiarízate con su aspecto.
- Sitúa el ratón sobre cada uno de sus elementos para conocer su significado.
- Abre las opciones de configuración de la consola y prueba sus diferentes opciones.
- Lanza en el *prompt* las instrucciones JavaScript que se te ocurran y comprueba los resultados.

2.1.4. Sentencias

Cuando se escribe código se hace a base de sentencias o instrucciones. Puede considerarse una sentencia como cada uno de los bloques de un Lego que se utiliza para construir el programa. Somos los programadores los que, teniendo un amplio conocimiento de cada estructura, las combinamos para obtener una aplicación que responda a nuestros intereses.

Un programa, por tanto, no es más que una sucesión de sentencias.

En JavaScript las sentencias se finalizan con el símbolo de punto y coma, aunque en algunos casos no es obligatorio ponerlo. Sin embargo, terminar cada instrucción con su correspondiente punto y coma es una norma de estilo muy extendida que tiene múltiples beneficios. Así, se mejora la legibilidad del programa y se minimiza la aparición de errores.

Cada una de las líneas de este fragmento de código es una sentencia:

```
let colores = {"rojo", "verde", "azul"};
contador++;
var primero = vector[0];
```

Un poco más adelante se verá el significado de cada una de ellas.

2.1.5. Bloques

Aunque las sentencias vistas hasta ahora son entidades independientes que se ejecutan de forma atómica, lo habitual es encontrarlas agrupadas en bloques.

Un bloque, por tanto, no es más que una secuencia de instrucciones encerradas entre llaves. Por ejemplo:

```
if (!cerrado) {
    abierto = 1;
    console.log("El candado está abierto");
}
```

Es posible ir un poco más allá y crear bloques anidados en varios niveles. Será muy frecuente encontrar estructuras como estas:

```

if (abierto) {
    console.log("Inicio del recorrido");
    for (iteracion = 0; iteracion < 5; iteracion++) {
        console.log(`Pasada número ${iteracion+1}`);
        if (iteracion == 4) {
            console.log("Fin del recorrido");
            abierto = 0;
        }
    }
}

```

En general, pueden entenderse los programas como una secuencia de bloques delimitados por llaves. Además, los bloques están íntimamente relacionados con la visibilidad, el alcance o el ámbito (scope) de otros elementos del lenguaje, como las variables.

2.1.6. Identificadores

A lo largo del programa va a ser necesario nombrar elementos y asignar nombres a diferentes estructuras de datos que permitan utilizarlas de acuerdo con las necesidades. Esos nombres se denominan identificadores. Por ejemplo, si se desea crear un espacio reservado en memoria que almacene la fecha del día en curso, se puede crear el identificador **fecha_actual**.

Además de los identificadores creados por el programador, todos los lenguajes de programación tienen una lista de identificadores predefinidos que solo se pueden usar para la utilidad que fueron diseñados y que existen para que se pueda programar. Son identificadores especiales llamados **palabras reservadas**. Estas son algunas de ellas: **break, case, const, let, try, void, while...**

En el fragmento de código del Apartado 2.1.4 **colores, contador, primero y vector** son identificadores definidos por el programador, y **let** y **var** son palabras reservadas del lenguaje.

Para saber más

Con este enlace o con el código QR se accede a la lista completa de palabras reservadas del lenguaje JavaScript que actualmente forman parte del estándar, así como a la lista de palabras reservadas que ya han dejado de serlo, y las que se prevén adoptar en un futuro cercano:

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Lexical_grammar#palabras_clave



Para definir un identificador hay que asegurarse de no violar ninguna de las siguientes reglas:

- Debe empezar obligatoriamente por una letra, guion bajo (_) o el símbolo del dólar (\$).
- Pueden seguirle más letras, números o guiones bajos.
- Distingue entre mayúsculas y minúsculas.
- Puede usar todas las letras que están definidas en UNICODE.

Recuerda

Es un error muy común no tener en cuenta que JavaScript distingue entre mayúsculas y minúsculas en los identificadores. No hay que olvidar que **Mielemento**, **miElemento**, **Mielemento** y **MIELEMENTO** son cuatro identificadores completamente distintos para JavaScript.



2.1.7. Comentarios

A lo largo del proceso de desarrollo de aplicaciones en entorno cliente va a surgir la necesidad de dejar comentarios en el código. Se trata de anotaciones, recordatorios o apuntes que se dejan en el código para aumentar la información sobre su significado. Los comentarios no alteran el flujo de ejecución del programa ni intervienen en la lógica de las aplicaciones.

Hay tres tipos de comentarios en JavaScript: comentarios de una línea, comentarios de varias líneas y comentarios *hashbang*.

En el primer caso se utilizan los caracteres **//** para iniciar el comentario. Se considerará que forma parte del comentario todo lo que aparezca a continuación y hasta final de la línea.

```

// Controla el primer bucle (número de iteraciones)
var contador = 0;
// Indica si el recurso está libre (true) u ocupado (false)
var recurso = true;

```

En el segundo caso, los comentarios de varias líneas o comentarios en bloque se inicián con **/*** y finalizan con ***/**. Se utilizan para realizar comentarios más extensos que mejoren el orden y la legibilidad del código.

```

/*
Almacena la cantidad de elementos actuales de la estructura.
Solo se incluyen los positivos.
Debe coincidir con los dispositivos válidos.
*/
let numElementos = 0;

```



Recuerda

Muchos de los errores relacionados con los comentarios se producen cuando se intentan anidar unos comentarios dentro de otros, una práctica que no está permitida con los comentarios de varias líneas. El siguiente fragmento de código produciría un error:

```

/*
Este comentario devolvería */
un /* syntax error por haber anidado
 */ comentarios de varias líneas
sin respetar sus reglas de uso
*/
var saludo = "Buenas tardes.";

```

El tercer y último tipo de comentario es el denominado **hashbang**. Se utiliza menos que los anteriores; son comentarios especiales y se usan principalmente para indicar la ruta a un motor de JavaScript específico que debe ejecutar el script.

Estos comentarios funcionan igual que los comentarios de una línea, comienzan con `#!` y solo son válidos al iniciar un script o módulo.

Este es su aspecto:

```
#!/usr/bin/env node
```

Recuerda



Para que los comentarios hashbang funcionen correctamente hay que asegurarse de que no existe ningún carácter, ni siquiera espacios en blanco, antes de `#!`. No son pocos los errores que se producen al descuidar esta regla.

2.2. Variables

Quizá sea el concepto más importante de esta unidad, puesto que las variables son el recurso indispensable más utilizado por los programadores.

Una variable es una zona de memoria a la que se asigna un nombre y en la que se guarda un valor. De esta forma, puede hacerse referencia a esa zona de memoria mediante un identificador que el programador ha definido, recuperar su valor o modificarlo.

2.2.1. Tipos de datos

Las variables son plenamente funcionales cuando se les asocia un valor, y todos los valores pertenecen a un tipo concreto. En JavaScript existe el tipo de dato **Object** y siete tipos de datos primitivos adicionales. Los siete tipos primitivos son: **string**, **number**, **boolean**, **undefined**, **null**, **bigint** y **symbol**. A continuación se analizan en detalle los primeros seis tipos primitivos, y el resto se verán más adelante, ya que trabajan con estructuras más complejas.

String

Un **string** no es más que una secuencia de caracteres que se utiliza para representar el texto. Hay tres formas de escribir una cadena de texto en JavaScript:

```
miString_1 = "con comillas dobles";
miString_2 = 'con comillas simples';
miString_3 = `con comillas invertidas o backticks`;
```

Las comillas simples y dobles permiten incluir a cada una de ellas cuando se usa la otra, como por ejemplo en:

```
miCadena_1 = "Mejor tiempo: 7";
miCadena_2 = 'Lanzado "Enginr", el nuevo motor eléctrico.';
```

Además, puede utilizarse el operador de concatenación (+) para unir fragmentos de cadenas o incluir el contenido de unas variables dentro de ellas:

```
subcadena_1 = "Primera parte, ";
subcadena_2 = "parte concatenada";
subcadena_3 = ". Última parte añadida";
cadenaFinal = "Cadena: " + subcadena_1 + subcadena_2 + subcadena_3 + ":";
```

En este ejemplo **cadenaFinal** contendría la cadena:

"Cadena: Primera parte, parte concatenada. Última parte añadida. "

La tercera forma de expresar las cadenas de caracteres simplifica el proceso de concatenación de cadenas. Los **backticks** permiten incluir expresiones sin tener que utilizar el operador +. Para ello, se incluye la expresión dentro de las llaves de \${}.

Por ejemplo, podría obtenerse la misma **cadenaFinal** anterior al hacer esto:

```
cadenaFinal = `Cadena: ${subcadena_1}${subcadena_2}${subcadena_3}.`;
```

Además, al hacer referencia a expresión se está haciendo referencia a cualquier elemento que devuelva un valor, como por ejemplo esta forma de calcular el área de un triángulo:

```
base = 3;
altura = 22;
document.write(`El área de nuestro triángulo es: ${base*altura/2}`);
```

Para finalizar con los **strings** es importante señalar que existe un conjunto de caracteres que no se pueden escribir, pero que son fundamentales en algunas situaciones. Son las conocidas como **secuencias de escape**.

¿Cómo puede separarse una cadena de caracteres en varias líneas? ¿Y cómo se puede tabular un párrafo? ¿Y si es necesario introducir unas comillas dobles en una cadena delimitada por comillas dobles? La respuesta es la misma en todos los casos: usando secuencias de escape. En la Tabla 2.3 se recogen algunas de las más comunes.

Tabla 2.3. Lista de las secuencias de escape más utilizadas

Secuencia	Uso
\'	Comillas simples
\\"	Comillas dobles
\\\	Barra invertida o backslash
\b	Retroceso
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulador horizontal
\v	Tabulador vertical

Aquí se muestran algunos de los usos más comunes de las secuencias de escape:

```
console.log("Así \"escapamos\" comillas dobles.");
console.log('Así \'escapamos\' comillas simples.');
console.log("Así \t tabulamos un texto.");
console.log("Así escapamos la barra \\ invertida.");
console.log("Así incluimos un\nsalto de línea. ");
```

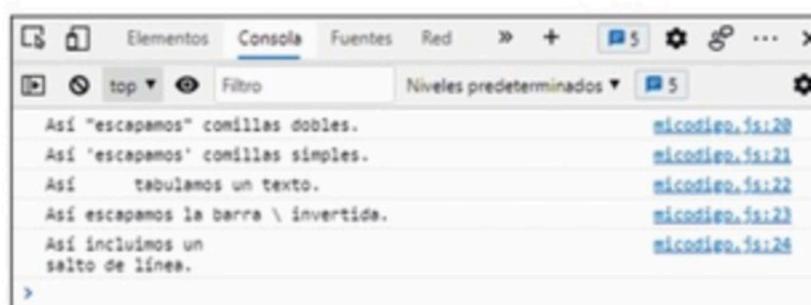


Figura 2.7. Salida por consola de las cadenas en las que se han incluido secuencias de escape.

Además, utilizando el patrón `\u{código}`, siendo **código** un valor hexadecimal de hasta ocho dígitos (32 bits), se puede representar cualquier carácter o símbolo de la tabla UNICODE. Así, si se quiere incluir en la cadena el símbolo del Bitcoin (฿) solo habría que escribir `\u{20BF}`.

Actividad propuesta 2.4

Uso de cadenas de caracteres

Vamos a comprobar si se han adquirido los conceptos básicos del trabajo con cadenas de caracteres:

Crea un fichero JavaScript que incluya una sola instrucción que sea capaz de sacar por consola la siguiente cadena:

El acceso a la ruta C:\\usuario\\ tarda 1'23", algo considerado "lento" en la actualidad.

Number

En JavaScript **Number** hace referencia a un único tipo de dato numérico que recoge cualquier formato de número con o sin decimales.

Para saber más

Los detalles del tipo de dato `number` vienen recogidos en la especificación IEEE 754 y viene representado por un formato de coma flotante de doble precisión de 64 bits. Con el siguiente enlace o con el código QR se puede encontrar toda la información relativa a este tipo de dato:

https://en.wikipedia.org/wiki/Double-precision_floating-point_format



© Ediciones Paraninfo

Para expresar un número, se escribe sin indicar ningún tipo de símbolo adicional. Si quiere indicarse que se trata de un número decimal se utiliza el punto (.) como separador decimal. Y si quiere indicarse un exponente se utiliza la e y el signo del exponente:

```
alturaCm = 182;
pesoKg = 71.6;
diametroTransistor = 2e-9;
```

Así, se estaría almacenando 182 cm, 71.6 kg y $2 \cdot 10^{-9}$ m.

Sin embargo, no puede darse por hecho que siempre se usará el sistema de representación decimal en los programas. Para indicarle al motor de JavaScript que se quiere operar en otro sistema de representación hay que escribir delante del número el dígito cero (0) seguido por un carácter de identificación, que informará al intérprete si se quiere trabajar en binario (b), octal (o) o hexadecimal (x).

Con estos ejemplos quedará más claro. Se trata de guardar el número 38 en otros sistemas de representación:

```
numeroBinario = 0b100110;
numeroOctal = 0o46;
numeroHexadecimal = 0x26;
console.log(`Binario: ${numeroBinario}\nOctal: ${numeroOctal}\nHexadecimal: ${numeroHexadecimal}`);
```

Como se aprecia, cada número lleva su propio «prefijo» para indicar que se trata de un sistema de representación distinto. Sin embargo, al observar su salida en consola se obtiene algo inesperado (Figura 2.8).

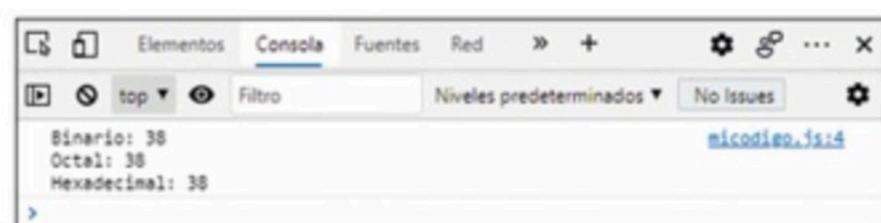


Figura 2.8. Salida por consola de los números en distintos sistemas de representación.

¿Qué ha ocurrido? Lo que ha sucedido es que la consola ha convertido todos los números al sistema decimal, puesto que está configurada para trabajar por defecto con ese sistema de representación numérica.

JavaScript tiene otra interesante capacidad numérica que no tienen la mayoría de los lenguajes de programación. A continuación se verá qué ocurre al trabajar con divisiones por cero:

```
divisionPorCero = 23/0;
otraOperacion = divisionPorCero + 12;
console.log(`División por cero: ${divisionPorCero}`);
console.log(`Otra operación: ${otraOperacion}`);
```

¡El lenguaje es capaz de entender y operar con el valor infinito! Toda una sorpresa si se tiene en cuenta que la gran mayoría de los lenguajes de programación lanzarían un error en ambos casos.

```
División por cero: Infinity
Otra operación: Infinity
```

Figura 2.9. Salida por consola del resultado de operaciones a priori no permitidas.

Para finalizar con el tipo de dato `number` hay que señalar otro comportamiento que, a diferencia de otros lenguajes de programación, también soporta JavaScript. ¿Qué ocurrirá al intentar operar con números y otros tipos de datos distintos?

```
operacion = 54 * "cadena de texto";
console.log(`Operación: ${operacion}`);
```

```
Operación: NaN
```

Figura 2.10. Resultado de operaciones no permitidas a los números.

Se obtiene como valor de salida `NaN`, que es el acrónimo de *Not a Number* (no es un número).

Que JavaScript, a diferencia de otros lenguajes de programación, considere tanto `Infinity` como `NaN` valores en vez de lanzar errores, no es una decisión meramente estética, sino que facilita notablemente la validación de errores y su consecuente localización.

Booleano

En informática en general, y en programación en particular, un booleano es un dato lógico que solo puede tomar dos valores: `true` (verdadero) o `false` (falso). Ambas son palabras reservadas del lenguaje que se pueden asignar:

```
valorVerdadero = true;
valorFalso = false;
```

Sin embargo, a la hora de evaluar expresiones lógicas se considera que todo lo que sea igual a cero es `false` y todo lo que sea distinto de cero es `verdadero`. Con algunas comprobaciones es posible aclarar el concepto usando la función `Boolean()`:

```
console.log(`1 - {true?: ${Boolean(true)}}`);
console.log(`2 - {false?: ${Boolean(false)}}`);
console.log(`3 - {1?: ${Boolean(1)}}`);
console.log(`4 - {0?: ${Boolean(0)}}`);
console.log(`5 - {"texto": ${Boolean("texto")}}`);
console.log(`6 - {"": ${Boolean("")}}`);
console.log(`7 - {Infinity?: ${Boolean(Infinity)}}`);
console.log(`8 - {NaN?: ${Boolean(NaN)}}`);
```

Expresión	Valor	Fuente
1 - {true?: true}	true	micodiego.js:1
2 - {false?: false}	false	micodiego.js:2
3 - {1?: true}	true	micodiego.js:3
4 - {0?: false}	false	micodiego.js:4
5 - {"texto": true}	true	micodiego.js:5
6 - {"": false}	false	micodiego.js:6
7 - {Infinity?: true}	true	micodiego.js:7
8 - {NaN?: false}	false	micodiego.js:8

Figura 2.11. Resultado de la evaluación lógica de las expresiones.

En sucesivos apartados se retomará el estudio de los booleanos ya que es preciso estar constantemente evaluando nuevas expresiones.

Undefined

En JavaScript se obtiene el valor `undefined` cuando todavía no se le ha asignado un valor a una variable y, en general, cuando no es capaz de evaluar una expresión. Más adelante, al estudiar estructuras más complejas, se volverán a ver nuevas características de este tipo de dato.

De momento, es suficiente con saber que `Boolean(undefined)` devuelve `false`.

Null

En el campo de la informática `null` siempre hace referencia a una dirección inválida, aunque su implementación práctica varía entre los diferentes lenguajes de programación. En JavaScript representa el valor vacío o nulo, de forma intencionada. Además, `Boolean(null)` es `false`.

Nota técnica

A los programadores principiantes de JavaScript les cuesta entender la diferencia entre `undefined` y `null`, puesto que ambos tipos de datos significan **ausencia de valor**. No obstante, existen matices que los convierten en tipos de datos completamente distintos. Este es el más importante: `undefined` significa que no hay valor porque aún no se ha definido; en cambio, `null` significa que no hay valor porque así lo ha indicado expresamente el programador.

BigInt

Cuando se trabaja con números muy grandes debe tenerse en cuenta que el tipo de dato `number`, tal como se conocía hasta ahora, podría crear algún problema.



El mayor y menor número que es capaz de manejar el tipo `number` se puede conocer usando las constantes `Number.MAX_VALUE` y `Number.MIN_VALUE`. Sin embargo, existe un rango de valores a partir del cual los enteros en JavaScript ya no son seguros y serán una aproximación de punto flotante de doble precisión del valor.

Por ello se recomienda que siempre que se vaya a usar un número menor que `Number.MIN_SAFE_INTEGER` o mayor que `Number.MAX_SAFE_INTEGER` se utilice el tipo de dato `BigInt` en lugar de `Number`.

Con `BigInt` se puede almacenar y operar de forma segura con números enteros muy grandes más allá del límite seguro de los enteros `number`. Para indicar al intérprete que se quiere usar el formato `BigInt` hay que finalizar el número con una `n`.

Jugando un poco con los valores puede conocerse su contenido y comprobar qué ocurre cuando se superan los límites:

```
console.log(`1. Límite superior de Number: ${Number.MAX_VALUE}`);
console.log(`2. Límite inferior de Number: ${Number.MIN_VALUE}`);
console.log(`3. Límite superior SEGURO: ${Number.MAX_SAFE_INTEGER}`);
console.log(`4. Límite inferior SEGURO: ${Number.MIN_SAFE_INTEGER}`);
console.log(`5. Superar el límite de Number: ${Number.MAX_VALUE+100}`);
console.log(`6. Superar el límite SEGURO de Number: ${9007199254740991+100}`);
console.log(`7. Superar el límite de SEGURO de Number usando BigInt: ${9007199254740991n + 100n}`);
```

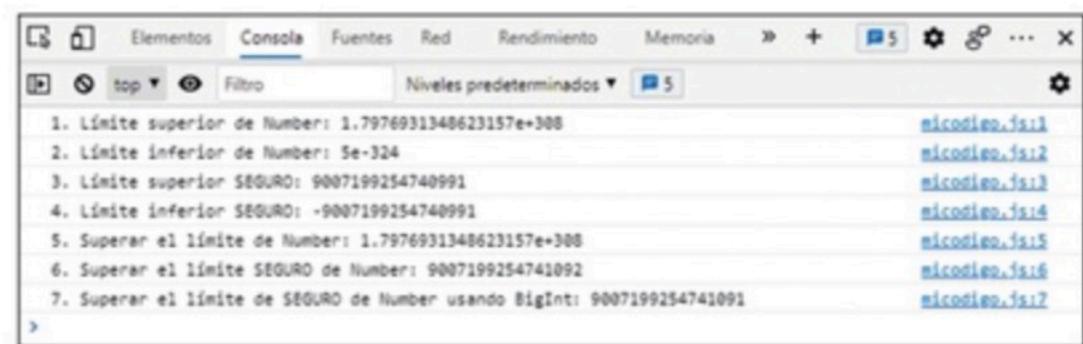


Figura 2.12. Resultados muy reveladores cuando se opera sobre los límites numéricos.

Hay algunos detalles interesantes. Teniendo en cuenta el resultado de la línea 1, ¿cómo es posible que al sumar 10 (línea 5) se obtenga el mismo resultado? La respuesta, como ya se supone, está en que se ha superado el límite superior del tipo de dato `Number` y por mucho que se sume no podrá obtenerse un número mayor.

Centrándose ahora en las líneas 3 y 5 se ve que la línea 3 devuelve el límite superior seguro de un `number`, y la línea 5 le suma 100 a ese límite. Sin embargo, al tomar los dos resultados y restarlos, ¡no se obtiene 100! Es otro problema con la precisión del tipo de dato `Number` al trabajar con cantidades muy grandes.

Por último, hay que ver lo que ocurre con el ejemplo anterior al forzar que los números acaben en `n`, es decir, cuando se trabaja con el tipo de dato `BigInt` en ese límite. Toman-do el valor de la línea 3 y convirtiéndolo en la línea 7 a `BigInt`, se le suma `100n` y ahora el

resultado sí que es correcto (la diferencia es exactamente 100). Se ha ganado en precisión numérica utilizando `BigInt` en vez de `Number`.

A veces se dan situaciones en las que los programadores no están seguros del tipo de dato con el que están trabajando. Para conocerlo puede utilizarse el operador `typeof`. Su funcionamiento es tan sencillo como el siguiente:

```
console.log('Tipo de true: ${typeof true}');
console.log('Tipo de 444: ${typeof 444}');
console.log('Tipo de 55.5: ${typeof 55.5}');
console.log('Tipo de 111n: ${typeof 111n}');
console.log('Tipo de "texto": ${typeof "texto"}');
console.log('Tipo de undefined: ${typeof undefined}');
console.log('Tipo de null: ${typeof null}');
console.log('Tipo de NaN: ${typeof NaN}');
console.log('Tipo de una variable no inicializada: ${typeof variable}');
```

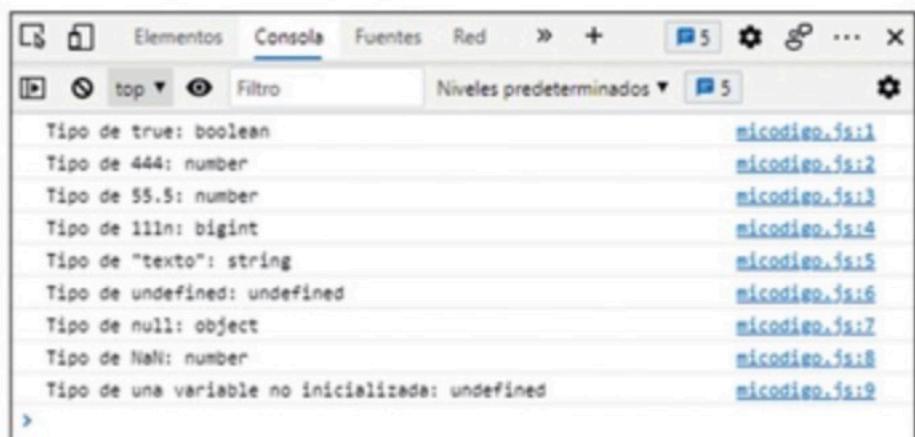


Figura 2.13. Los tipos de datos que se han visto hasta ahora usando el operador `typeof`.

Alguna interesante sorpresa se puede observar, como por ejemplo que el valor `NaN` (Not a Number) sea precisamente del tipo `Number`. Además, se observan dos nuevos conceptos:

1. El tipo de dato `Object`: es el más importante del lenguaje porque sobre él se construye toda la arquitectura de JavaScript. Se verá en profundidad más adelante.
2. Inicialización de variables: un concepto clave que se abordará en el siguiente apartado.

2.2. Conversión de tipos

Al inicio de esta unidad se decía que JavaScript llevaba en su ADN ser un lenguaje débilmente tipado. Esto significa que el lenguaje es capaz de lidiar con operaciones donde se mezclan tipos de datos distintos sin lanzar errores constantemente, tal y como ocurre en otros lenguajes de programación. Esto lo logra gracias a su potente capacidad para convertir unos tipos de datos en otros de forma dinámica.

Existen dos formas de realizar este tipo de conversiones: sin intervención del programador, o con la indicación explícita por parte del programador. A continuación se estudia cómo se realiza en cada caso.

Conversión automática de tipos

La regla de oro que aplica el lenguaje para realizar su conversión desasistida de tipos es actuar de la forma más razonable. Y es que, en ocasiones, al ver con qué inteligencia ha resuelto el lenguaje una operación, es fácil pensar: «claro, ha tomado el camino con más sentido».

Con todo y con eso, el lenguaje no es omnipotente, porque como se verá a continuación hay ocasiones en las que las conversiones sencillamente no son posibles.

```
console.log('1. true*7= ${true*7}');
console.log('2. 9-false= ${9-false}');
console.log('3. 12+"5"= ${12+"5"}');
console.log('4. "67"+11= ${"67"+11}');
console.log('5. 12*"5"= ${12*"5"}');
console.log('6. "67"*11= ${"67"*11}');
console.log('7. "texto"*8= ${"texto"*8}');
console.log('8. undefined/7= ${undefined/7}');
console.log('9. null*6= ${null*6}');
console.log('10. Infinity-"texto"= ${Infinity-"texto"}');
console.log('11. NaN+4= ${NaN+4}');
```

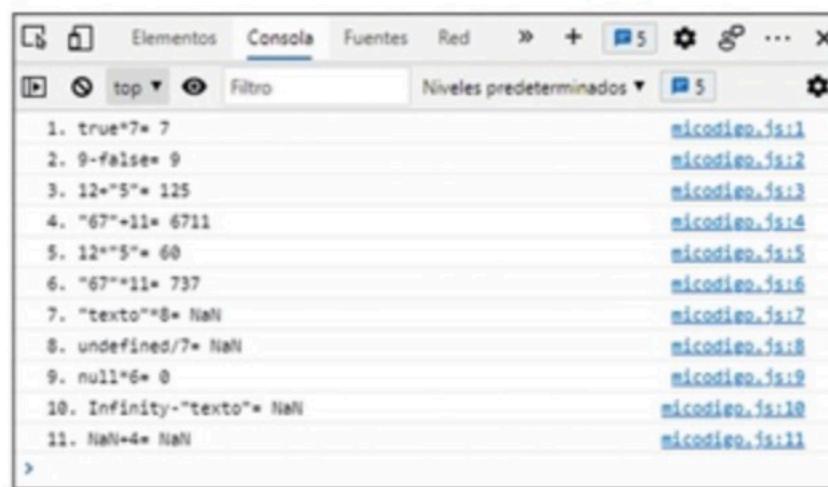


Figura 2.14. Conversiones automáticas de tipos sin intervención del programador.

En cada caso el lenguaje ha aplicado el sentido común de la siguiente forma:

- El valor booleano **true** se considera que es toda aquella expresión que se evalúa a distinto de cero. Desde el punto de vista numérico se entiende que **true** es 1 y **false** es 0. Por tanto, se sustituye **true** por 1 y se multiplica.
- Siguiendo el mismo razonamiento anterior, se sustituye **false** por 0 y se resta.

- Se intenta sumar dos números, pero uno de ellos está encerrado entre comillas dobles, por tanto, es un **string**. En el contexto de los **strings** se había estudiado que se utiliza el operador **+** para concatenar cadenas. Y esa es precisamente la lógica que se ha seguido, convertir el 12 a cadena de caracteres y concatenarlas.
- Siguiendo el mismo razonamiento anterior, se ha convertido el 11 en una cadena de caracteres y los ha concatenado.
- En este caso el operador es *****; por tanto, y a diferencia de los dos casos anteriores, solo cabría pensar que lo que se desea es multiplicar. Se convierte la cadena de texto "5" en el número 5 y se multiplica.
- Siguiendo el mismo razonamiento anterior, se convierte "67" en 67 y se multiplica.
- Una vez más, el mismo caso que los dos anteriores, salvo por el detalle de que la cadena de texto no es un número. No es posible realizar operaciones matemáticas cuando una de las partes no es un número. Y así se indica en el resultado.
- Siguiendo el mismo razonamiento anterior, no puede hacerse una operación matemática cuando una de las partes no está definida.
- Null** se había definido como la ausencia intencionada de valor por parte del programador. Parece razonable que se considere 0 cuando interviene en una operación matemática.
- Se trata del mismo caso que el explicado en el punto 7; se intenta operar un número muy grande con una cadena de caracteres.
- Resultado evidente, toda vez que se intenta realizar una operación matemática sabiendo que una de las partes indica explícitamente que no es un número.

Conversión manual de tipos

Podría surgir la necesidad de saltar el comportamiento predeterminado del lenguaje y forzar una conversión en un sentido distinto. ¿Cómo puede forzarse que en los ejemplos 3 y 4 el operador **+** funcione como una suma en lugar de como concatenador de cadenas de caracteres? Muy sencillo, convirtiendo explícitamente el **string** numérico al tipo **Number**:

```
console.log('3. 12+"5"= ${12+Number("5")}');
console.log('4. "67"+11= ${Number("67")+11}');
```

¿Y en caso de querer hacer justo lo contrario? De querer hacer que dos números en una suma se concaténen en vez de sumarse, existe la posibilidad de forzar su tipo a **string** de este modo:

```
console.log(`6+7 Sin forzar: ${6+7}`);
console.log(`6+7 Forzando: ${String(6)+String(7)}`);
```

Tras ejecutar la primera instrucción se obtendría un resultado de 13 (suma) y tras ejecutar la segunda el resultado sería "67" (concatenación de cadenas de caracteres). Este recurso de conversión se puede utilizar para cada cualquier tipo de dato existente en JavaScript.

Por tanto, como programadores hay dos estrategias que pueden seguirse para que las conversiones entre tipos distintos se realicen de acuerdo con las necesidades de cada momento:

1. Si se conoce en profundidad cómo funciona el intérprete y hay un 100 % de seguridad de qué camino tomará la conversión, se puede delegar la decisión en JavaScript.
2. Si no se dan las dos condiciones anteriores, lo mejor es forzar conversiones de forma explícita utilizando las herramientas que ofrece el lenguaje.

Dicho todo lo cual, sería interesante recordar que las reglas con las que opera el lenguaje son cambiantes a lo largo del tiempo, por lo que no es mala idea repasar las novedades del estándar cada cierto tiempo.

2.2.3. Declaración e inicialización

Una vez aprendido qué son las variables y de qué tipos de datos pueden ser, ha llegado el momento de crearlas (declararlas) y asignarles un valor por primera vez (inicializarlas).

Declarar una variable no es más que asignarle un identificador a una zona de memoria. Existen tres formas de declarar una variable en JavaScript, y es utilizando las palabras reservadas **var**, **let** y **const**, que se analizan a continuación.

Var

Es la forma tradicional de declarar una variable. Su uso es tan sencillo como este:

```
var miVariable;
```

Con la instrucción anterior se le indica al programa esto: «reserva un espacio en memoria y llámale **miVariable**».

```
miVariable = 8;
```

Ahora se le está indicando al programa: «toma el valor 8 y almacénalo en el espacio de memoria que me habías reservado y al que habíamos llamado **miVariable**». Es la primera vez que se asigna un valor a esta variable, es decir, se ha inicializado.

Solo es necesario declarar las variables una vez. Después pueden utilizarse tantas veces como se quiera simplemente usando su identificador.

El ámbito de uso de estas variables es el contexto de ejecución en el que se encuentra la palabra **var**. Al final del siguiente concepto se comprenderá mejor esto último.

Let

Es una forma de declarar variables que se permite desde la versión 6 de ECMAScript, y para entender su funcionamiento hay que ahondar en el concepto de ámbito de las variables, que se anunció en el apartado de los bloques.

La gran diferencia entre **let** y **var** es que el ámbito de **let** es más local que el de **var**. Se dice que el ámbito de las variables declaradas con **let** es el bloque. Se puede comprobar con el siguiente código:

```
{
  var miVariable = "Soy visible fuera del bloque.";
  let otraVariable = "No soy visible fuera del bloque.";
}
console.log(miVariable);
console.log(otraVariable);
```

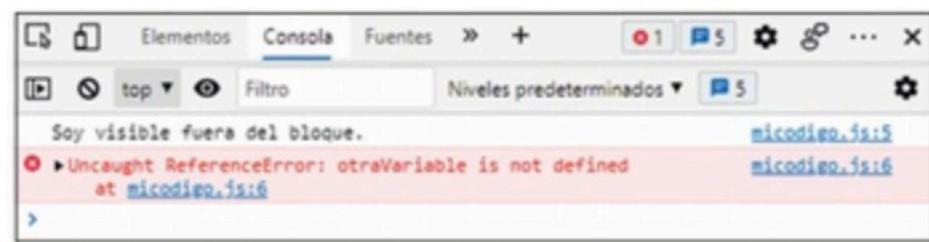


Figura 2.15. Error como consecuencia del ámbito de la variable definida con **let**.

A la vista de los resultados, puede comprobarse que la variable definida por **let** solo es accesible dentro del bloque donde se definió. Sin embargo, aquella definida con **var** sí que es accesible.

No obstante, esto no significa que el ámbito de una variable definida con **var** sea total, puesto que también sufre restricciones dependiendo del lugar donde se declare. Un ejemplo servirá de aclaración:

```
var miVariable = "He sido declarada fuera de todo bloque";
function unaFuncion() {
  var miVariable = "He sido declarada dentro de una función";
  return 0;
}
console.log(miVariable);
```

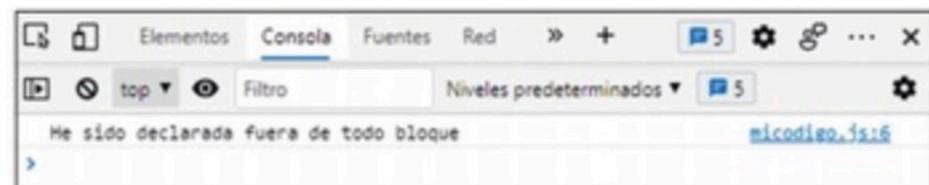


Figura 2.16. El intérprete selecciona el valor global de la variable.

A pesar de que la segunda declaración de la misma variable aparece más tarde, el intérprete saca el valor que se le asignó la primera vez. Es un revelador ejemplo de **visibilidad de una variable**. Realmente existen dos variables llamadas **miVariable**, una local que existe solo dentro de la función, y otra global que existe en el resto del archivo.

Const

En los dos tipos de variables que se acaban de ver pueden declararse e inicializarse las variables y posteriormente usarlas y modificar su contenido tantas veces como se considere oportuno. Pero ¿y con **const**?

```
const IVA = 0.21;
IVA = 0.18;
```

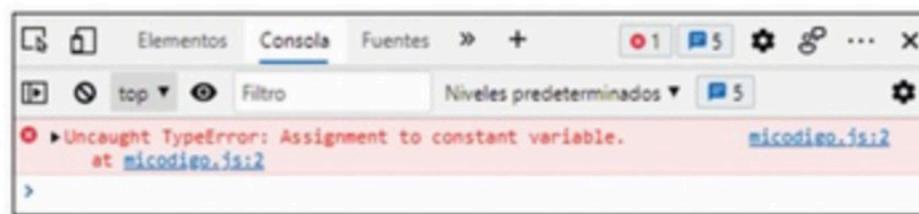


Figura 2.17. Error al intentar modificar el valor de una constante.

Efectivamente, con **const** se definen constantes, es decir, variables a las que se asigna un valor que no cambiará nunca. ¿Una variable que no varía? Exactamente.

Desde el punto de vista del ámbito, funciona igual que **let**. Además, y aunque no es obligatorio, es una norma de estilo con amplio consenso asignar a las constantes identificadores en mayúsculas para distinguirlas de los demás tipos de variables. Así, con un simple golpe de vista, se evita caer en la tentación de asignarles otro valor, y, en consecuencia, de lanzar un error.

2.3. Entrada y salida en el navegador

Para mejorar la comprensión de los conceptos vistos hasta ahora, se venían utilizando algunas de las funcionalidades que ofrecen los navegadores para sacar datos en pantalla. Pero no son las únicas, ni se han visto en profundidad. A continuación se estudia cómo proporcionar datos al navegador y también cómo obtener respuestas y otros mensajes de salida.

2.3.1. Mensajes en la consola

Un recurso muy habitual que utilizan los programadores durante el desarrollo y despliegue de sus aplicaciones es enviar texto a la consola. Es bastante útil porque permite obtener mucha información de depuración de los programas sin «ensuciar» el aspecto de esos programas y, al mismo tiempo, son fácilmente localizables para retirarlos antes de poner las aplicaciones en entornos de producción.

Existen cuatro tipos de mensajes que se pueden generar para sacarlos en la consola: **console.log()**, **console.info()**, **console.warn()** y **console.error()**. Todos ellos muestran un texto con una estética diferente (que puede cambiar según el navegador) y se pueden utilizar los filtros que proporciona el navegador para ver solo aquellos del tipo que interese.

Este es el aspecto que presentan:

```
console.log("Versión LOG de un mensaje por consola.");
console.info("%cVersión INFO del mensaje número %d por consola.", "font-weight:bold;", 2);
console.warn("Versión WARN de un mensaje por consola.");
console.error("Versión ERROR de un mensaje por consola.");
```

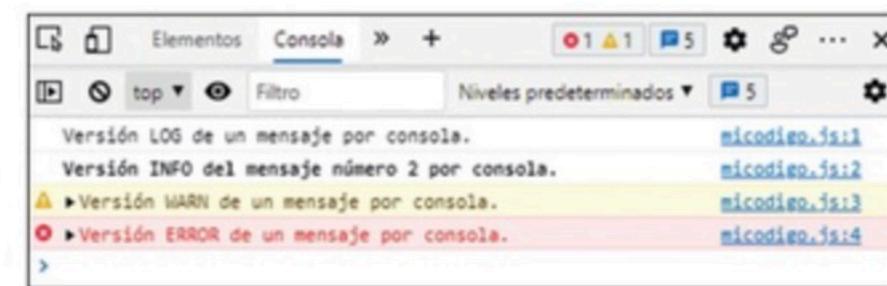


Figura 2.18. Uso y decoración de mensajes generados para la consola.

Como puede verse, cada tipo de mensaje se muestra con la apariencia típica de mensajes de *log*, errores o *warnings*. Los dos primeros mensajes se generan con la misma estética; sin embargo, es frecuente que se utilicen sustituciones de cadena para enriquecer el resultado.

Si se observa el segundo mensaje, al comenzar una cadena por **%c** puede establecerse un estilo CSS personalizado. Además, si en cualquier parte de la cadena se coloca **%d** o **%s**, puede sustituirse dinámicamente por un valor numérico o de cadena de caracteres respectivamente.

En la Tabla 2.4 se recogen otros métodos útiles que pueden lanzarse en la consola para obtener información de interés sobre los programas y su ejecución en el navegador.

Tabla 2.4. Lista adicional de métodos útiles para trabajar con la consola

Método	Descripción
console.dir()	Muestra un listado interactivo de las propiedades de un objeto JavaScript específico.
console.group()	Crea un nuevo grupo que tabula los mensajes de la consola. Para retroceder un nivel de tabulación se utiliza groupEnd() .
console.groupCollapsed()	Igual que group() pero aparecen todos los mensajes colapsados en uno, siendo necesario pulsar un botón para desplegarlos.
console.table()	Muestra los datos en forma de tabla.
console.time()	Inicia un temporizador con un nombre personalizado (pueden crearse hasta 10 000 simultáneamente en la misma página). El temporizador se detiene con timeEnd() .
console.trace()	Muestra una traza del error.

2.3.2. Mensajes de confirmación

En este caso se trata de un cuadro de diálogo que puede lanzarse con `confirm()` y un mensaje con dos opciones: **Aceptar** o **Cancelar**. Si el usuario pulsa el botón de aceptar se recibirá `true` como respuesta y `false` si decide pulsar el botón de cancelar. Así de simple. El ejemplo siguiente lo muestra:

```
let mensaje = "¿Estás seguro de querer eliminar?";
let respuesta = confirm(mensaje);
console.log(`Respuesta del cuadro de diálogo: ${respuesta}.`);
```



Figura 2.19. Cuadro de confirmación tras ejecutar la utilidad `confirm()`.



Figura 2.20. Valor devuelto por el cuadro de confirmación tras pulsar Aceptar.

2.3.3. Mensajes de entrada

Existe una variante al cuadro de diálogo anterior que además de mostrar un mensaje y dos botones para aceptar y rechazar la acción propuesta, proporciona una caja de texto para que el usuario pueda escribir. Esto se hace a través de la herramienta `prompt()`.

Si el usuario pulsa el botón **Cancelar** se obtiene `null` en la variable que controla la respuesta. Si el usuario pulsa el botón **Aceptar** pero no ha escrito nada, se obtiene una cadena vacía. Si escribe algo y pulsa **Aceptar**, se recibirá la cadena introducida.

```
let mensaje = "Para eliminar escriba ELIMINAR";
let respuesta = prompt(mensaje);
console.log(`El usuario escribió: ${respuesta}.`);
```



Figura 2.21. Cuadro de entrada de texto relleno por el usuario.

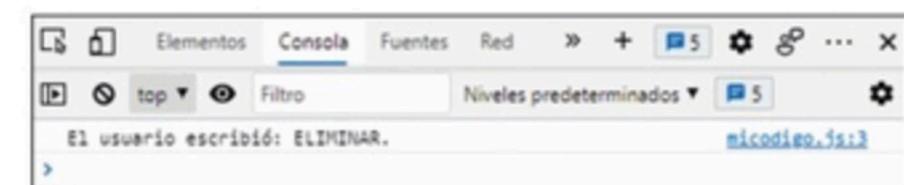


Figura 2.22. Salida del valor devuelto por el cuadro de texto tras pulsar Aceptar.

Además, puede establecerse un valor predeterminado que aparecerá dentro de la caja de texto. En ocasiones es útil para dirigir al usuario. Se consigue pasando un segundo parámetro a `prompt()`.

```
let mensaje = "¿Qué IVA desea aplicar?";
let respuesta = prompt(mensaje, "21%");
console.log(respuesta);
```



Figura 2.23. Cuadro de entrada de texto con valor por defecto.

2.3.4. Mensajes de alerta

El último tipo de mensaje que se verá ya se adelantó en alguna ocasión. Se trata simplemente de lanzar una pequeña ventana donde se mostrará algún mensaje de interés para el usuario. Para cerrarlo basta con pulsar el botón **Aceptar**:

```
const PI = 3.14159;
alert(`Recuerda usar esta aproximación de π en tus cálculos: ${PI}`);
```

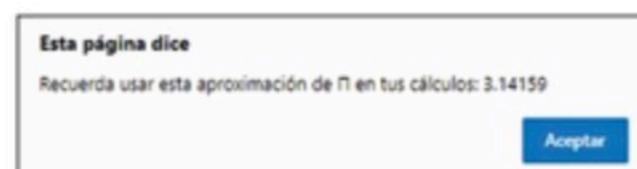


Figura 2.24. Mensaje de alerta en Microsoft Edge.

Importante

Si bien la utilización de mensajes de alerta, cuadros de confirmación y cuadros de entrada de texto son muy útiles desde el punto de vista del aprendizaje, es importante destacar que son cosas del pasado. En la actualidad ninguna aplicación web sería utilizar estas herramientas del navegador para interactuar con el usuario. Existen multitud de soluciones mucho más profesionales para este cometido.



Actividad propuesta 2.5

Mensajes del navegador

Combina todos los mensajes del navegador vistos hasta ahora. Para ello, crea un fichero JavaScript que tras ejecutarlo en el navegador:

- Pida el nombre al usuario. El usuario deberá introducir su nombre.
- Le pregunte al usuario si quiere abandonar el programa. El usuario deberá pulsar **Aceptar** o **Cancelar**.
- Lance una alerta con la decisión del usuario.
- Muestre en la consola «FIN DEL PROGRAMA». El mensaje debe aparecer en negrita, subrayado y en color azul.

2.4. Operadores

Los operadores son símbolos que forman parte de la identidad del lenguaje, que permiten manipular los operandos, es decir, indican la operación que se va a realizar con los datos que intervienen.

Es importante destacar que el intérprete de JavaScript evalúa la mayoría de los operadores de izquierda a derecha y algunos de derecha a izquierda. El orden en el que se evalúan sigue un orden de precedencia que se estudiará al final de este apartado.

Tal y como ocurre en las matemáticas, pueden utilizarse los paréntesis para alterar el orden de precedencia y forzar la evaluación ordenada de las partes que interesen.

A continuación se aborda el estudio de la amplia variedad de operadores de JavaScript.

2.4.1. Operadores de asignación

Asignan al operando izquierdo el valor del operando derecho por medio del símbolo `=`. En la expresión `a = b` se toma el valor de `b` y se asigna a `a`. También se dispone de otras versiones de asignaciones compuestas, como las mostradas en la Tabla 2.5.

Tabla 2.5. Lista completa con los operadores de asignación compuesta de JavaScript

Nombre	Aplicación	Simplificación
Asignación	<code>a = b</code>	<code>a = b</code>
Asignación de adición	<code>a = a + b</code>	<code>a += b</code>
Asignación de resta	<code>a = a - b</code>	<code>a -= b</code>
Asignación de multiplicación	<code>a = a * b</code>	<code>a *= b</code>
Asignación de división	<code>a = a / b</code>	<code>a /= b</code>
Asignación de módulo	<code>a = a % b</code>	<code>a %= b</code>

Nombre	Aplicación	Simplificación
Asignación de exponenciación	<code>a = a ** b</code>	<code>a **= b</code>
Asignación de desplazamiento a la izquierda	<code>a = a << b</code>	<code>a <<= b</code>
Asignación de desplazamiento a la derecha	<code>a = a >> b</code>	<code>a >>= b</code>
Asignación de desplazamiento a la derecha sin signo	<code>a = a >>> b</code>	<code>a >>>= b</code>
Asignación AND bit a bit	<code>a = a & b</code>	<code>a &= b</code>
Asignación XOR bit a bit	<code>a = a ^ b</code>	<code>a ^= b</code>
Asignación OR bit a bit	<code>a = a b</code>	<code>a = b</code>
Asignación AND lógico	<code>a = a && (a = b)</code>	<code>a &&= b</code>
Asignación OR lógico	<code>a = a (a = b)</code>	<code>a = b</code>
Asignación de anulación lógica	<code>a = a ?? (a = b)</code>	<code>a ??= b</code>

En los fragmentos de código de la Tabla 2.5 hay numerosos conceptos que probablemente todavía no se entienden, pero será por poco tiempo. De momento, es suficiente con dominar la asignación y el patrón que se sigue al simplificar las operaciones en las que intervienen.

2.4.2. Operadores de comparación

Se llaman así porque su función es determinar si una comparación entre dos operandos es verdadera (`true`) o falsa (`false`). El resultado de la operación, por tanto, es siempre un valor booleano.

Los operandos pueden ser de diverso tipo: numéricos, cadenas de caracteres, booleanos u objetos. Las cadenas de caracteres se comparan atendiendo al orden alfabético del lugar que ocupan sus representaciones UNICODE.

Recuerda

Hay que tener en cuenta que a pesar de que en nuestro idioma la «ñ» se encuentra en la posición alfabética número 15, dentro de UNICODE ocupa la posición 241. Por tanto, desde el punto de vista de la comparación, la «ñ» será el mayor de los caracteres del alfabeto español.



Actividad propuesta 2.6

UNICODE

Para conocer la tabla de símbolos UNICODE es necesario repasarla varias veces:

- Abre tu navegador y busca la tabla de caracteres UNICODE en internet.
- Recorre toda la tabla y fíjate el lugar numérico que ocupan caracteres como la «ñ» y la «Ñ» en relación con el resto de las letras de nuestro alfabeto.
- Revisa la tabla completa para que descubras la cantidad de símbolos que tienes a tu disposición.

En la gran mayoría de los casos, cuando dos operandos no son del mismo tipo, el intérprete intenta convertirlos a tipos compatibles para la comparación. Generalmente, se terminan comparando valores numéricos (salvo en casos como `==` y `!=`).

En la relación de la Tabla 2.6 se muestra qué resultado se obtiene al aplicar la variedad de operadores de comparación de JavaScript.

Tabla 2.6. Operadores de comparación y su significado

Operador	Significado	Resultado true
Igualdad (<code>==</code>)	Devuelve <code>true</code> si los operandos son iguales.	<code>5 == 5</code> <code>"5" == 5</code> <code>5 == '5'</code>
Distinto (<code>!=</code>)	Devuelve <code>true</code> si los operandos no son iguales.	<code>5 != 9</code> <code>9 != "5"</code>
Igualdad estricta (<code>==</code>)	Devuelve <code>true</code> si los operandos son iguales y del mismo tipo.	<code>5 === 5</code>
Desigualdad estricta (<code>!=</code>)	Devuelve <code>true</code> si los operandos son del mismo tipo pero no iguales, o son de diferente tipo.	<code>5 !== "5"</code> <code>5 !== '5'</code>
Mayor que (<code>></code>)	Devuelve <code>true</code> si el operando izquierdo es mayor que el operando derecho.	<code>9 > 5</code> <code>"11" > 9</code>
Mayor o igual que (<code>>=</code>)	Devuelve <code>true</code> si el operando izquierdo es mayor o igual que el operando derecho.	<code>9 >= 3</code> <code>9 >= 9</code>
Menor que (<code><</code>)	Devuelve <code>true</code> si el operando izquierdo es menor que el operando derecho.	<code>3 < 9</code> <code>"9" < 11</code>
Menor o igual que (<code><=</code>)	Devuelve <code>true</code> si el operando izquierdo es menor o igual que el operando derecho.	<code>5 <= 9</code> <code>9 <= 9</code>

Para saber más

En muchas ocasiones, los programadores principiantes suelen usar la notación `=>` cuando intentan expresar «mayor o igual», en lugar de usar la forma correcta `>=`. Es un error peligroso porque el símbolo `=>` tiene otro significado muy importante en JavaScript y por ello a veces el error es indetectable.



2.4.3. Operadores aritméticos

Son los clásicos de las matemáticas que todos conocen. Toma los valores numéricos de los operandos y les aplica la operación aritmética que indique el operador. El resultado es también numérico.

Los cuatro operadores básicos son suma (+), resta (-), multiplicación (*) y división (/). Además de los anteriores, JavaScript proporciona otros seis operadores aritméticos, de los cuales los cuatro primeros son muy utilizados (Tabla 2.7).

Tabla 2.7. Operadores aritméticos y su significado

Operador	Significado	Ejemplo
Módulo (%)	Devuelve el resto de la división entera.	<code>9 % 5</code> devuelve 4
Incremento (++)	Operador unario que suma uno a su operando. Preincremento (<code>++a</code>): incrementa el valor de <code>a</code> en 1 y devuelve su contenido. Posincremento (<code>a++</code>): devuelve su contenido y después incrementa el valor de <code>a</code> en 1.	<code>let a = 9;</code> <code>b = ++a;</code> <code>b</code> vale 10 <code>let a = 9;</code> <code>b = a++;</code> <code>b</code> vale 9
Decremento (--)	Operador unario que resta uno a su operando. Predecremento (<code>--a</code>): decremente el valor de <code>a</code> en 1 y devuelve su contenido. Posdecremento (<code>a--</code>): devuelve su contenido y después decremente el valor de <code>a</code> en 1.	<code>let a = 9;</code> <code>b = --a;</code> <code>b</code> vale 8 <code>let a = 9;</code> <code>b = a--;</code> <code>b</code> vale 9
Potencia (**)	Calcula la potencia de un número elevando el operando izquierdo al exponente del operando derecho.	<code>2 ** 3</code> devuelve 8
Negativo unario (-)	Devuelve la negación de su operando.	Si <code>a</code> es 5 entonces devuelve -5
Positivo unario (+)	Intenta convertir el operando en un número, si aún no lo es.	<code>+3</code> devuelve 3 <code>+true</code> devuelve 1

Actividad propuesta 2.7

Operadores aritméticos

Crea un fichero JavaScript y lanza por consola todas las operaciones de ejemplo de la tabla anterior. Modifica sus valores e intenta adivinar el resultado antes de ejecutarlo. ¿Has obtenido los valores esperados?

2.4.4. Operadores bit a bit

Los operadores bit a bit tratan a sus operandos como una ristra de 32 bits. Es decir, operan en binario bit a bit, en lugar de hacerlo en decimal, octal o hexadecimal. Una vez terminada la operación, el intérprete devolverá el resultado nuevamente convertido a decimal.

En la Tabla 2.8 se muestra un resumen de los operadores bit a bit que JavaScript pone a disposición del programador.

En la gran mayoría de los casos, cuando dos operandos no son del mismo tipo, el intérprete intenta convertirlos a tipos compatibles para la comparación. Generalmente, se terminan comparando valores numéricos (salvo en casos como `==` y `!=`).

En la relación de la Tabla 2.6 se muestra qué resultado se obtiene al aplicar la variedad de operadores de comparación de JavaScript.

Tabla 2.6. Operadores de comparación y su significado

Operador	Significado	Resultado true
Igualdad (<code>==</code>)	Devuelve <code>true</code> si los operandos son iguales.	<code>5 == 5</code> <code>"5" == 5</code> <code>5 == '5'</code>
Distinto (<code>!=</code>)	Devuelve <code>true</code> si los operandos no son iguales.	<code>5 != 9</code> <code>9 != "5"</code>
Igualdad estricta (<code>==</code>)	Devuelve <code>true</code> si los operandos son iguales y del mismo tipo.	<code>5 === 5</code>
Desigualdad estricta (<code>!=</code>)	Devuelve <code>true</code> si los operandos son del mismo tipo pero no iguales, o son de diferente tipo.	<code>5 !== "5"</code> <code>5 !== '5'</code>
Mayor que (<code>></code>)	Devuelve <code>true</code> si el operando izquierdo es mayor que el operando derecho.	<code>9 > 5</code> <code>"11" > 9</code>
Mayor o igual que (<code>>=</code>)	Devuelve <code>true</code> si el operando izquierdo es mayor o igual que el operando derecho.	<code>9 >= 3</code> <code>9 >= 9</code>
Menor que (<code><</code>)	Devuelve <code>true</code> si el operando izquierdo es menor que el operando derecho.	<code>3 < 9</code> <code>"9" < 11</code>
Menor o igual que (<code><=</code>)	Devuelve <code>true</code> si el operando izquierdo es menor o igual que el operando derecho.	<code>5 <= 9</code> <code>9 <= 9</code>

Para saber más

En muchas ocasiones, los programadores principiantes suelen usar la notación `=>` cuando intentan expresar «mayor o igual», en lugar de usar la forma correcta `>=`. Es un error peligroso porque el símbolo `=>` tiene otro significado muy importante en JavaScript y por ello a veces el error es indetectable.



2.4.3. Operadores aritméticos

Son los clásicos de las matemáticas que todos conocen. Toma los valores numéricos de los operandos y les aplica la operación aritmética que indique el operador. El resultado es también numérico.

Los cuatro operadores básicos son suma (+), resta (-), multiplicación (*) y división (/). Además de los anteriores, JavaScript proporciona otros seis operadores aritméticos, de los cuales los cuatro primeros son muy utilizados (Tabla 2.7).

Tabla 2.7. Operadores aritméticos y su significado

Operador	Significado	Ejemplo
Módulo (%)	Devuelve el resto de la división entera.	<code>9 % 5</code> devuelve 4
Incremento (++)	Operador unario que suma uno a su operando. Preincremento (<code>++a</code>): incrementa el valor de <code>a</code> en 1 y devuelve su contenido. Posincremento (<code>a++</code>): devuelve su contenido y después incrementa el valor de <code>a</code> en 1.	<code>let a = 9;</code> <code>b = ++a; b vale 10</code> <code>let a = 9;</code> <code>b = a++; b vale 9</code>
Decremento (--)	Operador unario que resta uno a su operando. Predecremento (<code>--a</code>): decremente el valor de <code>a</code> en 1 y devuelve su contenido. Posdecremento (<code>a--</code>): devuelve su contenido y después decremente el valor de <code>a</code> en 1.	<code>let a = 9;</code> <code>b = --a; b vale 8</code> <code>let a = 9;</code> <code>b = a--; b vale 9</code>
Potencia (**)	Calcula la potencia de un número elevando el operando izquierdo al exponente del operando derecho.	<code>2 ** 3</code> devuelve 8
Negativo unario (-)	Devuelve la negación de su operando.	Si <code>a</code> es 5 entonces devuelve -5
Positivo unario (+)	Intenta convertir el operando en un número, si aún no lo es.	<code>+3</code> devuelve 3 <code>+true</code> devuelve 1

Actividad propuesta 2.7

Operadores aritméticos

Crea un fichero JavaScript y lanza por consola todas las operaciones de ejemplo de la tabla anterior. Modifica sus valores e intenta adivinar el resultado antes de ejecutarlo. ¿Has obtenido los valores esperados?

2.4.4. Operadores bit a bit

Los operadores bit a bit tratan a sus operandos como una ristra de 32 bits. Es decir, operan en binario bit a bit, en lugar de hacerlo en decimal, octal o hexadecimal. Una vez terminada la operación, el intérprete devolverá el resultado nuevamente convertido a decimal.

En la Tabla 2.8 se muestra un resumen de los operadores bit a bit que JavaScript pone a disposición del programador.

Tabla 2.8. Operadores bit a bit y su significado

Operador	Uso	Descripción
AND a nivel de bits	<code>a & b</code>	Devuelve un 1 en cada posición del bit donde ambos operandos son 1.
OR a nivel de bits	<code>a b</code>	Devuelve un 0 en cada posición del bit donde ambos operandos son 0.
XOR a nivel de bits	<code>a ^ b</code>	Devuelve un 0 en cada posición del bit donde ambos operandos son iguales, o 1 si ambos operandos son distintos.
NOT a nivel de bits	<code>~ a</code>	Invierte los bits de su operando.
Desplazamiento a la izquierda	<code>a << b</code>	Desplaza el primer operando <code>a</code> a la izquierda tantos bits como indique el operando <code>b</code> . Los bits en exceso desplazados a la izquierda se descartan. Los bits cero se desplazan desde la derecha.
Desplazamiento a la derecha (con propagación de signo)	<code>a >> b</code>	Desplaza el primer operando <code>a</code> a la derecha tantos bits como indique el operando <code>b</code> . Los bits en exceso desplazados a la derecha se descartan. Las copias del bit más a la izquierda se desplazan desde la izquierda. Dado que el nuevo bit más a la izquierda tiene el mismo valor que el bit anterior a la izquierda, el bit de signo (el bit más a la izquierda) no cambia.
Desplazamiento a la derecha sin signo (o desplazamiento a la derecha de relleno cero)	<code>a >>> b</code>	Desplaza a la derecha el primer operando el número de bits indicado. Los bits en exceso desplazados hacia la derecha se descartan. Los bits cero se desplazan desde la izquierda. El bit de signo se convierte en 0, por lo que el resultado nunca es positivo. A diferencia de los otros operadores a nivel de bits, el desplazamiento a la derecha de relleno cero devuelve un entero de 32 bits sin signo.

Son unos operadores que se utilizan en muy contadas ocasiones, así que no es necesario ahondar mucho más en ellos.

2.4.5. Operadores lógicos

Son uno de los tipos de operadores más importantes de cualquier lenguaje de programación puesto que son la forma de evaluar condiciones que permitirán tomar decisiones a lo largo del flujo de ejecución del programa.

Se utilizan normalmente con valores booleanos y devuelven otro valor booleano. Sin embargo, conceptualmente devuelven el valor de uno de los operandos que intervienen

en la operación. En consecuencia, si se utilizan con operandos no booleanos, podrían devolver un valor no booleano. Son los que se recogen en la Tabla 2.9.

Tabla 2.9. Operadores lógicos

Operador	Uso	Descripción
AND lógico (<code>&&</code>)	<code>expr1 && expr2</code>	Devuelve <code>expr1</code> si se puede convertir a <code>false</code> , de lo contrario devuelve <code>expr2</code> . Por tanto, cuando se usa con valores booleanos, devuelve <code>true</code> si ambos operandos son <code>true</code> y <code>false</code> en caso contrario.
OR lógico (<code> </code>)	<code>expr1 expr2</code>	Devuelve <code>expr1</code> si se puede convertir a <code>true</code> , de lo contrario devuelve <code>expr2</code> . Por tanto, cuando se usa con valores booleanos, devuelve <code>true</code> si alguno de los operandos es <code>true</code> o <code>false</code> si ambos son <code>false</code> .
NOT lógico (<code>!</code>)	<code>!expr</code>	Devuelve <code>false</code> si su único operando se puede convertir a <code>true</code> ; en caso contrario devuelve <code>true</code> .

Las expresiones que pueden convertirse a `false` son aquellas que se evalúan a `0`, "", `null`, `NaN` o `undefined`.

Para tener una comprensión completa de qué valor se puede esperar al evaluar expresiones donde intervienen los operadores lógicos AND (`&&`), OR (`||`) y NOT(`!`), puede resultar útil echar un vistazo a las tablas de verdad donde un 1 representa `true` y un 0 representa `false` (Tabla 2.10).

Tabla 2.10. Tablas de verdad de los operadores lógicos

<code>a</code>	<code>b</code>	<code>a && b</code>	<code>a</code>	<code>b</code>	<code>a b</code>	<code>a</code>	<code>!a</code>
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1	0	1
1	1	1	1	1	1	0	1

Ahora es el momento de observar qué resultados ofrece JavaScript tras ejecutar estas operaciones:

```
// OPERACIONES AND
console.log(`1 = ${true && true}`);
console.log(`2 = ${true && false}`);
console.log(`3 = ${false && true}`);
console.log(`4 = ${false && (3 == 4)}`);
console.log(`5 = {'Gato' && 'Perro'}`);
console.log(`6 = ${false && 'Gato'}`);
console.log(`7 = {'Gato' && false}`);
// OPERACIONES OR
console.log(`8 = ${true || true}`);
```

```
console.log(`9 = ${false || true}`);
console.log(`10 = ${true || false}`);
console.log(`11 = ${false || (3 == 4)}`);
console.log(`12 = ${'Gato' || 'Perro'}`);
console.log(`13 = ${false || 'Gato'}`);
console.log(`14 = ${'Gato' || false}`);
// OPERACIONES NOT
console.log(`15 = ${!true}`);
console.log(`16 = ${!false}`);
console.log(`17 = ${!`Gato'}`);
```

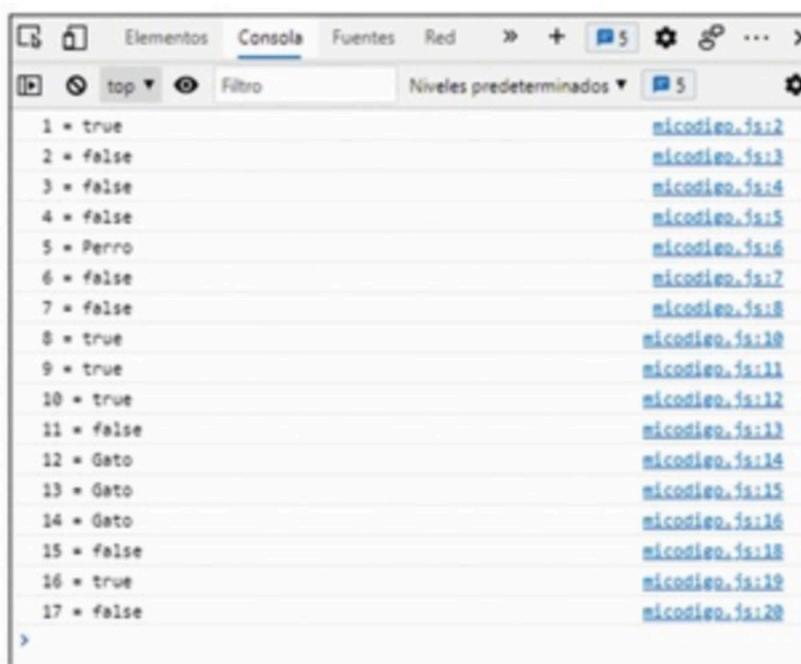


Figura 2.25. Resultado de la ejecución de las instrucciones con operadores lógicos ofrecidas como ejemplo.

Sabiendo que en las operaciones lógicas las expresiones se evalúan de izquierda a derecha, existe un truco que utilizan mucho los programadores para comprobar que el flujo de ejecución del programa discurre por los cauces esperados. Son las **evaluaciones de cortocircuito**, y se construyen así:

- **false && cualquier cosa** se evalúa en cortocircuito siempre como **false**.
- **true || cualquier cosa** se evalúa en cortocircuito siempre como **true**.

Las reglas de los operadores lógicos garantizan que estas evaluaciones sean siempre correctas. Esto es así porque la parte derecha, **cualquier cosa**, nunca se llega a evaluar, por lo que se garantiza que el resultado sea el que se coloca a la izquierda del operador.

Hay que tener en cuenta, además, que para el segundo caso puede usarse un operador aprobado muy recientemente, el **operador de fusión nulo** (**??**) que funciona como **||** pero solo devuelve la segunda expresión cuando la primera es **null**.

2.4.6. Operadores de cadena

En varias ocasiones se ha utilizado ya el operador de cadena. Es la versión del símbolo **+** que se utiliza para concatenar cadenas de caracteres, tal y como se vio al inicio de esta unidad.

```
console.log("Así" + " concatenamos " + 4 + " cadenas"); // "Así concatenamos 4 cadenas"
```

Hay que fijarse muy bien en cómo se utilizan los espacios dentro de la segunda y cuarta cadenas para que la frase resultante sea coherente. De no hacerlo así, el resultado habría sido «Asíconcatenamos4cadenas».

También resulta muy útil el operador de asignación simplificado con la concatenación:

```
let cadena = "nombre";
cadena += " y apellidos";
console.log(cadena); // "nombre y apellidos"
```

2.4.7. Operador condicional

Al operador condicional también se le suele llamar operador ternario, puesto que es el único de JavaScript en el que intervienen tres operandos. Resulta ser de enorme utilidad en una amplia variedad de situaciones por su simplicidad y el ahorro de líneas de código que proporciona. La sintaxis es: **condición ? valor1 : valor2**.

Si la condición se evalúa a **true**, la operación devuelve **valor1**; en caso contrario devuelve **valor2**.

```
let edad = 21;
let mensaje = (edad >= 18) ? "mayor" : "menor";
console.log(`El usuario es ${mensaje} de edad.`); // "El usuario es mayor de edad."
```

Como ya se habrá podido adivinar, la variable **mensaje** toma el valor **"mayor"** cuando la variable **edad** es mayor o igual que 18; en caso contrario toma el valor **"menor"**. Por último, solo resta sustituir el valor de la variable **mensaje** al sacar el texto por consola.

2.4.8. Precedencia de operadores

Se hace referencia a la precedencia de operadores cuando se desea conocer el orden en el que se aplican al evaluar una expresión. Tal y como ocurre en las matemáticas, pueden utilizarse paréntesis para forzar la ruptura de esta precedencia y evaluar en primer lugar aquellas expresiones que interesen.

En la Tabla 2.11 se muestra esa precedencia, de mayor a menor. En la tabla se muestran operadores aún desconocidos porque operan con tipos y estructuras de datos más complejas que se verán más adelante.

Tabla 2.11. Precedencia de los operadores

Tipo de operador	Operadores
Miembro	. []
Llamar / crear instancia	() new
Negación / incremento	! ~ - + ++ -- typeof void delete
Multiplicar / dividir	* / %
Adición / sustracción	+ -
Desplazamiento bit a bit	<< >> >>>
Relacional	< <= > >= in instanceof
Igualdad	== != === !==
AND bit a bit	&
XOR bit a bit	^
OR bit a bit	
AND lógico	&&
OR lógico	
Condicional	?:
Asignación	= += -= *= /= %= <<= >>= >>>= &= ^= = &&= = ??=
Coma	,

2.5. Estructuras de control

Hasta este momento, para construir un programa con las instrucciones que ya se conocían se tenía la limitación de ejecutarlas secuencialmente, una tras otra hasta el final. Es más, por muy grande que se haga un programa siempre se sabe qué se va a ejecutar y qué no, con un simple golpe de vista.

A pesar de que esta forma de programar ya ofrece alguna ventaja de automatización, está bastante limitada. Es preciso disponer de otras instrucciones que permitan elegir alternativas de ejecución, plantear bifurcaciones o realizar tareas repetitivas en un punto dado antes de continuar con la siguiente instrucción. Dicho de otra forma, que sea imposible determinar con un simple golpe de vista cuál será el camino que tomará la ejecución de un programa en función de sus datos de entrada.

JavaScript cuenta con todas esas instrucciones. De hecho, es uno de los fundamentos del diseño de algoritmos en la totalidad de los lenguajes de programación. Son las denominadas **estructuras de control**.

Las estructuras de control son instrucciones que permiten alterar el flujo de ejecución de un programa, de manera que habrá líneas de código que se ejecuten y otras que no. El elemento que determinará esas trayectorias es la **condición**, normalmente construida con expresiones lógicas. Si una condición se cumple, la ejecución del programa tomará un camino, y si no se cumple tomará otro.

Las estructuras de control se clasifican en tres grandes grupos que se estudiarán a continuación: estructuras de control selectivas, repetitivas y de salto.

2.5.1. Estructuras de control selectivas

Las estructuras de control selectivas se llaman así porque, en función de una condición, permiten elegir un camino u otro. En el programa se tiene una lógica establecida y se quiere que ejecute unas instrucciones cuando se cumpla una condición, u otras cuando no se cumpla.

Siguiendo este patrón de comportamiento se van a estudiar dos estructuras de control selectivas: la sentencia **if** y la sentencia **switch**.

if

Es la estructura de control selectiva más simple. Su funcionamiento básico es sencillo: si se cumple la condición se ejecutan las instrucciones que contiene, y si no se cumple, se ejecutan las instrucciones del bloque **else** (si está presente). Esta es su sintaxis:

```
if (condición) {
    instrucciones_si_true;
}
else {
    instrucciones_si_false;
}
```

Importante

En las estructuras **if**, y en general en todas las estructuras que utilicen bloques, si el contenido del bloque lo forma una única instrucción, no es necesario escribir las llaves.

Por ejemplo:

```
console.log("Inicio");
let local = 2;
let visitante = 1;
if (local == visitante) {
    console.log("¡Hay empate!");
}
else {
    console.log("¡NO hay empate!");
}
console.log("Fin");
```



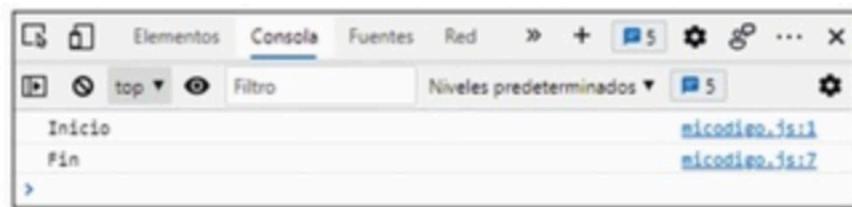


Figura 2.26. Ejecución de una sentencia if else.

Se observa cómo se ejecuta el interior del segundo bloque como consecuencia de haberse evaluado a **false** la condición.

Ahora que ya se ha aprendido a discriminar entre dos opciones, ¿por qué no tomar decisiones en función de un número de opciones deseadas? Es el momento de presentar a **if elseif else**:

```
if (condición1) {
    instrucciones_si_condición1_true;
}
else if (condición2) {
    instrucciones_si_condición2_true;
}
...
else if (condiciónN) {
    instrucciones_si_condiciónN_true;
}
...
else {
    instrucciones_si_todas_condiciones_false;
}
```

En esta variante se ejecutará cada bloque correspondiente si su condición se evalúa a **true**. Si todas ellas se evalúan a **false**, entonces se ejecuta el bloque de instrucciones del **else**. A continuación se verá con un ejemplo:

```
console.log("Inicio");
let local = 2;
let visitante = 1;
if (local > visitante) {
    console.log("Local gana.");
}
else if (local < visitante) {
    console.log("Visitante gana");
}
else {
    console.log("¡Hay empate!");
}
console.log("Fin");
```

Obsérvese cómo se ha construido esta estructura de tres partes. Primero se comprueba si gana el local y después si gana el visitante. La única opción que queda si nada de lo anterior ha ocurrido, es que hayan empatado, por lo que no es necesario establecer ninguna condición y resulta útil el **else**. Por supuesto, si alguna de las dos primeras condiciones se cumple, no se ejecutarían ninguna de las otras dos.

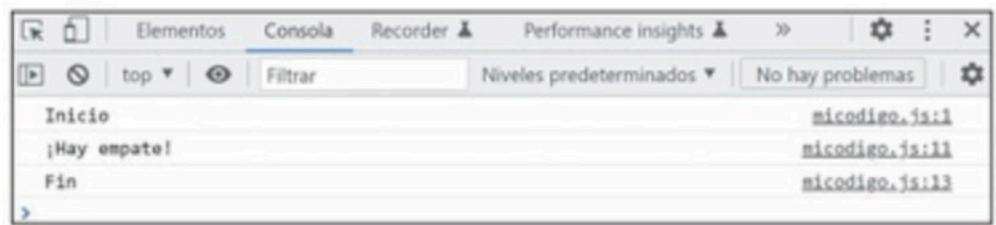


Figura 2.27. Ejecución de una sentencia if elseif else.

Pero todavía no se han agotado las capacidades de la sentencia **if**, y es que, como casi todas las estructuras de programación, ¡pueden anidarse! Se puede incluir una estructura dentro de otra cuando la complejidad de las opciones de decisión así lo requieran. Véase este último ejemplo:

```
console.log("Inicio");
let local = 2;
let visitante = 1;
if (local > visitante) {
    console.log("Local gana.");
    if ((local-visitante) > 1) {
        console.log("Y además por goleada.");
    }
    else {
        console.log("Pero por la mínima.");
    }
}
else if (local < visitante) {
    console.log("Visitante gana");
    if ((visitante-local) > 1) {
        console.log("Y además por goleada.");
    }
    else {
        console.log("Pero por la mínima.");
    }
}
else {
    console.log("¡Hay empate!");
}
console.log("Fin");
```

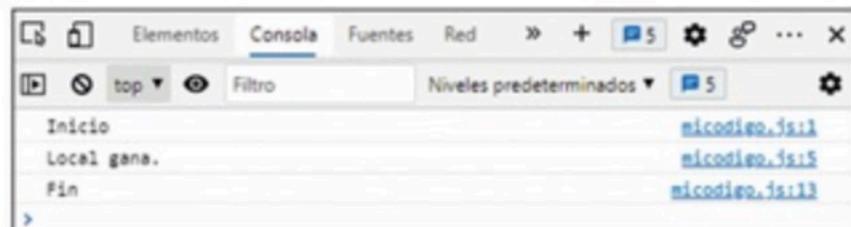


Figura 2.28. Ejecución de una sentencia if elseif else anidada.

Se ha incluido otra estructura **if else** dentro de cada uno de los dos bloques donde no hay empate. Quiere saberse si la diferencia de goles ha sido amplia o no. En la condición

anidada se restan los goles de los equipos y se pregunta si es mayor que 1. Si es mayor que 1 se considera una goleada y, si no, la victoria es por la mínima. Da igual cuál de los dos equipos gane, el programa seguirá ofreciendo correctamente la información.

Importante

Una de las claves que forman parte de las normas de estilo entre los programadores es estructurar correctamente el código aplicando tabuladores y espacios. La legibilidad y comprensión del código aumentan notablemente cuando se tiene en cuenta.



Actividad propuesta 2.8

Estructura de control if

Crea un fichero JavaScript e incluye las instrucciones necesarias que te permitan ejecutar la siguiente lógica:

- Pídele su nombre y edad al usuario.
- Utiliza la estructura de control if para determinar en qué franja de edad se encuentra, de manera que consideres lo siguiente:
 - Niño: hasta 12 años.
 - Adolescente: de 13 a 17 años.
 - Trabajador: de 18 a 64 años.
 - Jubilado: de 65 años o más.
- El programa debe terminar mostrando este tipo de mensaje en verde y negrita por consola: «Juan tiene 21 años y por tanto es Trabajador».

Switch

La sentencia **switch** cobra sentido cuando tras evaluar una expresión cuyo resultado puede ser variado, es preciso abrir múltiples vías distintas de ejecución. Esta es su sintaxis:

```
switch (expresión) {
  case valor_1:
    instrucciones_1
    [break;]
  case valor_2:
    instrucciones_2
    [break;]
  ...
  default:
    instrucciones_predeterminadas
    [break;]
}
```

Tras evaluar **expresión** se ejecutan las instrucciones del bloque **case** cuyo valor coincide con el resultado de la evaluación. Si el valor de ningún **case** coincide, entonces se ejecuta la sección **default**. Las instrucciones **break** aparecen entre corchetes porque son opcionales. Si se pone, ahí terminará la ejecución del **switch**. Si no se pone, se ejecuta

también «todo» lo que hay desde ahí hasta el final, o hasta que se encuentre un **break**. En el ejemplo siguiente se ve su funcionamiento:

```
console.log("Menú abierto");
let letra_pulsada = 'c';
switch (letra_pulsada) {
  case 'a':
    console.log("Abrir archivo");
    break;
  case 'c':
    console.log("Copiar");
    break;
  case 'p':
    console.log("Pegar");
    break;
  default:
    console.log("Opción incorrecta");
}
console.log("Menú cerrado");
```

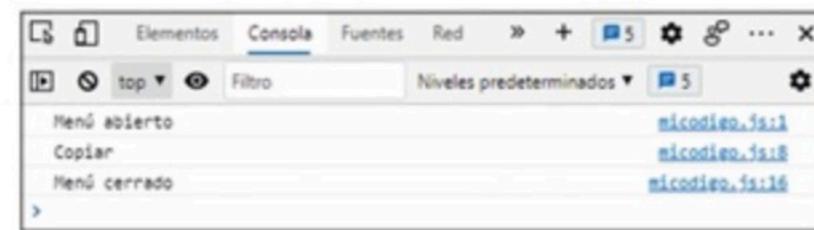


Figura 2.29. Ejecución de una sentencia switch.

De no haber puesto el **break**, habría ocurrido lo que se muestra en la Figura 2.30.

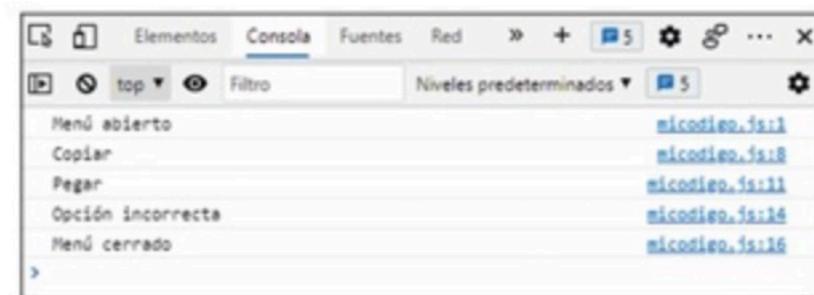


Figura 2.30. Ejecución de una sentencia switch sin break.

Actividad propuesta 2.9

Estructura de control switch

Crea un fichero JavaScript e incluye las instrucciones necesarias que te permitan ejecutar la siguiente lógica de gestión de horarios de apertura:

- Pídele al usuario la inicial del día de la semana (L, M, X, J, V, S, D).
- Por medio de la estructura de control **switch** muestra por consola el horario de apertura del día elegido. Por supuesto, el horario debe ser diferente cada día.

2.5.2. Estructuras repetitivas

Las estructuras repetitivas vienen a cubrir otra importante necesidad de los algoritmos: la ejecución repetitiva de una o varias instrucciones. Ya no es el caso de querer bifurcar o establecer distintos caminos de ejecución, sino hacerlo de forma repetitiva tantas veces como indique una condición específica.

Dentro de estas estructuras se van a estudiar los bucles **while**, **do while** y **for**.

While

El bucle **while** ejecuta las instrucciones que formen parte de su interior mientras que su condición se evalúe a **true**. Solo terminará cuando la condición resulte **false**. Por tanto, las instrucciones que forman parte del bucle se ejecutarán 0, 1 o más de una vez. Esta es su sintaxis:

```
while (condición) {
    instrucciones;
}
```

Es preciso fijarse en el comportamiento del bucle porque hay algo muy importante que no puede olvidarse:

```
let pases = 0;
while (pases < 10) {
    console.log('Pase número ${pases+1}');
    pases++;
}
```

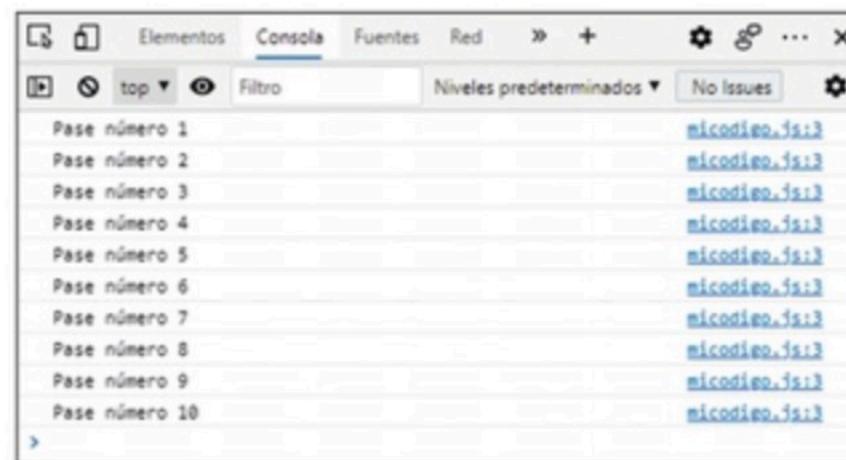


Figura 2.31. Ejecución de un bucle while.

Asegurarse de que la variable que controla la condición de continuación se modifique dentro del bucle es crucial. De no hacerlo, es muy probable que nunca se dé una condición de salida y se produzca un bucle infinito.

Recuerda

 Las consecuencias de que se produzca un bucle infinito en un programa van mucho más allá de los problemas derivados de que el programa deje de funcionar. Provocará un consumo de recursos desproporcionado, puede colgar el navegador e incluso la propia máquina en la que se ejecuta.

Actividad propuesta 2.10

Estructura de control while

Vamos a jugar a adivinar la letra.

- Crea un programa con un bucle **while** cuya condición de salida sea el valor '**s**'.
- Dentro del bucle **while** pídele al usuario una letra.
- Si la letra elegida es la «**s**» el programa finaliza y muestra un mensaje de éxito por consola. Si la letra elegida no es la «**s**» se muestra un mensaje por consola de no ha habido suerte y se le vuelve a pedir una letra al usuario.
- El bucle se ejecutará indefinidamente hasta que el usuario introduzca la letra «**s**».

Do while

El bucle **do while** es una variante del bucle **while**. La condición se sitúa al final del bucle, en lugar del principio. Y ahí reside su única diferencia con respecto a **while**; el bucle **do while** se ejecuta siempre al menos una vez. En el resto de sus comportamientos son idénticos.

Su sintaxis es la siguiente:

```
do {
    instrucciones;
} while (condición);
```

A continuación se muestra un ejemplo de este bucle:

```
console.log("--- Primeros 10 números pares ---");
let contador = 0;
let numero = 1;
do {
    if (numero%2 == 0) {
        console.log(`PAR: ${numero}`);
        contador++;
    }
    numero++;
} while (contador<10);
```

Tal y como se ha indicado, en esta ocasión la condición del bucle **do while**, que decide si se sigue iterando o no, se sitúa al final de la estructura. Es importante tener esto siempre muy presente para elegir correctamente el bucle necesario en cada caso.

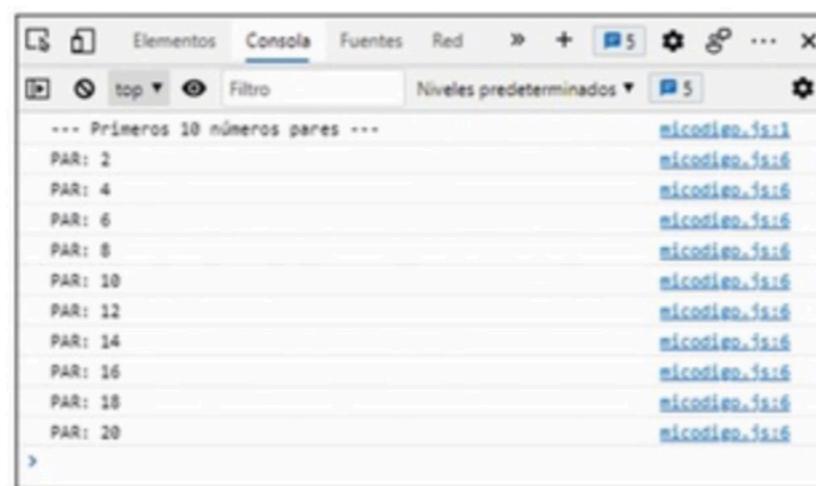


Figura 2.32. Ejecución de un bucle do while.

Con este programa se muestran los 10 primeros números pares. Se necesitan dos variables, una para controlar la condición de salida, es decir, cuándo se tienen 10 números (**contador**); y otra para ir buscando los números pares desde el 1 en adelante (**número**). Se inicia el bucle y se comprueba con un **if** si el resto de la división entera del número actual (1) entre 2 da cero (condición indispensable para ser par). Si da cero, el número es par, se visualiza por consola y se suma 1 al **contador** de números pares. A continuación, se incrementa el valor de **número** para probar con el siguiente. Finalmente se comprueba en la condición de salida del bucle si ya se tienen los 10 pares (para salir del bucle) o si no se tienen aún (para continuar iterando).

Actividad propuesta 2.11

Estructura de control do while

Modifica el código de tu Actividad propuesta 2.10 y sustituye el bucle **while** por un bucle **do while**. Comprueba que todo sigue funcionando correctamente.

For

El bucle **for** es otra estructura de control repetitiva muy importante, que se ejecuta una y otra vez hasta que la condición especificada se evalúe a **false**. Es una de las sentencias que más se utilizan desarrollando aplicaciones web del lado cliente. Su sintaxis es la siguiente:

```
for ([expresiónInicial]; [expresiónCondicional]; [expresiónDeActualización]) {
    instrucciones;
}
```

Esto es lo que ocurre cuando se ejecuta:

1. Se ejecuta la inicialización por medio de **expresiónInicial** (si existe, porque es opcional). Su función es iniciar uno o varios contadores, aunque técnicamente permite la inclusión de cualquier expresión.

2. Se evalúa **expresiónCondicional** para saber si el bucle continúa o debe finalizar. Si el resultado es **true**, se ejecutan las instrucciones del bucle (paso 3). Si es **false**, el bucle termina. También es posible no indicar esta expresión, en cuyo caso siempre se considerará **true** y el bucle se ejecutará sin fin hasta que otra instrucción lo rompa.
3. Se ejecuta el bloque de **instrucciones**.
4. Se ejecuta **expresiónDeActualización**, normalmente para actualizar la variable que controla la condición de salida. Su presencia también es opcional.
5. El flujo de ejecución del programa vuelve al paso 2.

A modo de ejemplo se tomará un bucle que sacará por consola cualquier tabla de multiplicar que se le indique:

```

const TABLA = 9;
for (let contador=1; contador<=10; contador++) {
    console.log(`${TABLA} x ${contador} = ${TABLA*contador}`);
}
  
```

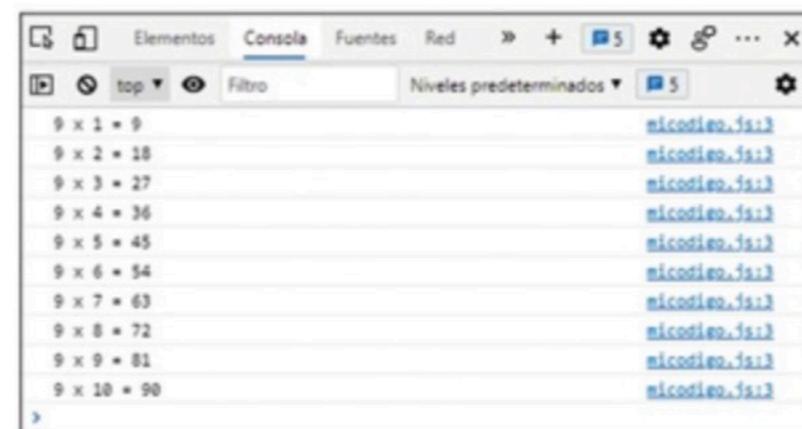


Figura 2.33. Ejecución de un bucle for.

Existen más variantes de bucles **for** en JavaScript que se estudiarán tan pronto como se conozcan las estructuras de datos que utilizan.

Actividad propuesta 2.12

Estructura de control for

Crea un programa que muestre en la consola todos los números múltiplos de 7 que hay del 1 al 100. Al finalizar debe indicar cuántos son.

2.5.3. Estructuras de salto

Las instrucciones de salto son un interesante recurso que forma parte de la caja de herramientas de los programadores cuando se trabaja con bucles. Como se verá a continuación, permiten romper el flujo de ejecución establecido por los bucles cuando se dan situaciones excepcionales que quieren controlarse.

Para saber más

Las instrucciones de salto son muy útiles en situaciones muy concretas, pero no debe abusarse de ellas. Un exceso de instrucciones de salto termina convirtiendo el programa en **código espagueti**, o lo que es lo mismo, un galimatías en el que nadie es capaz de seguir el flujo de ejecución del programa.

A continuación se explica cómo funcionan cada una de las instrucciones: **break**, **continue** y la declaración **labeled**.

Break

La instrucción **break** se utiliza para terminar un bucle **while**, **do while** o **for** o una sentencia **switch** y transferir el control a la siguiente instrucción.

Cuando se estudió la sentencia **switch** se vio cómo la sentencia **break** conseguía romper la secuencia de ejecución para que no se ejecutaran el resto de los **case**. Pues, en general, se usa para romper el bucle envolvente más cercano a él.

Su efecto puede verse con claridad modificando el último ejemplo:

```
const TABLA = 9;
for (let contador=1; contador<=10; contador++) {
    console.log(`TABLA ${TABLA} x ${contador} = ${TABLA*contador}`);
    if (contador == 5)
        break;
}
```

Al ejecutarlo se ve cómo ahora solo se muestra la mitad de la tabla de multiplicar. Esto ocurre por haber introducido una nueva condición: si **contador** es 5 (si se ha llegado a la mitad), **break**, es decir, rompe el bucle.

Continue

A veces, estando en medio de una iteración de un bucle, se desea descartar el resto de las instrucciones del bloque y volver a evaluar la condición. Exactamente esa es la función de **continue**.

Al utilizar **continue** se finaliza la iteración actual de un bucle **while**, **do while** o **for** y continúa la ejecución del bucle con la siguiente iteración (en el bucle **for** se ejecuta también **expresiónDeActualización**).

Mejor verlo en funcionamiento:

```
console.log("Primeros 10 impares NO x 3");
let contador = 0;
let numero = 1;
while (contador < 10){
    if (numero%3 == 0) {
        numero++;
        continue;
    }
    console.log(`IMPAR: ${numero}`);
    contador++;
    numero++;
}
```

```
}
if (numero%2 != 0) {
    console.log(`IMPAR: ${numero}`);
    contador++;
}
numero++;
```

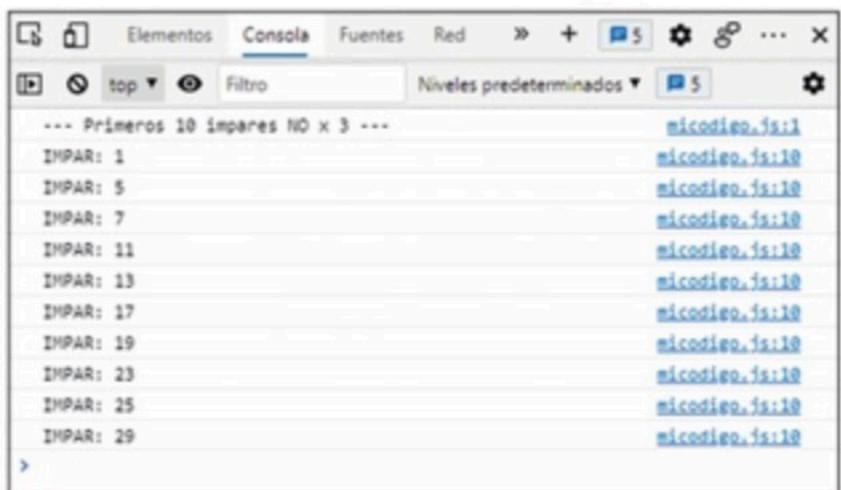


Figura 2.34. Ejecución de una sentencia **continue**.

El programa anterior calcula los 10 primeros números impares que no son múltiplo de 3. Siguiendo con la estrategia para el cálculo de los 10 primeros números pares que se desarrolló en el estudio del bucle **do while**, se modifica la condición para detectar los números impares. Además, para cumplir con el segundo requisito del programa (que el número no sea múltiplo de 3) se añade un **if** previo que calcula precisamente eso usando el operador módulo (resto de la división entera). Si el módulo 3 de un número es cero, se trata de un múltiplo de 3. Se invoca entonces a **continue** para que no siga ejecutando el resto del bucle y vuelve a evaluarse la condición principal, la del **while**.

Labeled

La declaración **labeled** viene a complementar la funcionalidad de las dos instrucciones anteriores. Su utilidad reside en establecer puntos en el programa, a los que se asigna un nombre (una etiqueta) al que hacer referencia cuando se desea efectuar un salto.

De esta forma, ya no solo se puede romper un bucle o continuar con la condición de evaluación principal, sino que existe la capacidad de saltar a cualquier punto del programa, previamente etiquetado.

Así, puede usarse **break etiqueta**; para romper el bucle etiquetado con ese identificador, o **continue etiqueta**; para descartar el resto del bucle y evaluar la condición principal del bucle etiquetado.

El siguiente ejemplo muestra el uso de **labeled** con **break**:

```
let primero = segundo = 1;
buclePrincipal: while (true) {
    console.log(`Bucle principal) Iteración ${primero}`);
    primero++;
    while (true) {
        console.log(`(Bucle secundario) Iteración ${segundo}`);
        segundo++;
        if (segundo == 5)
            break;
        else if (primero == 3)
            break buclePrincipal;
    }
}
```

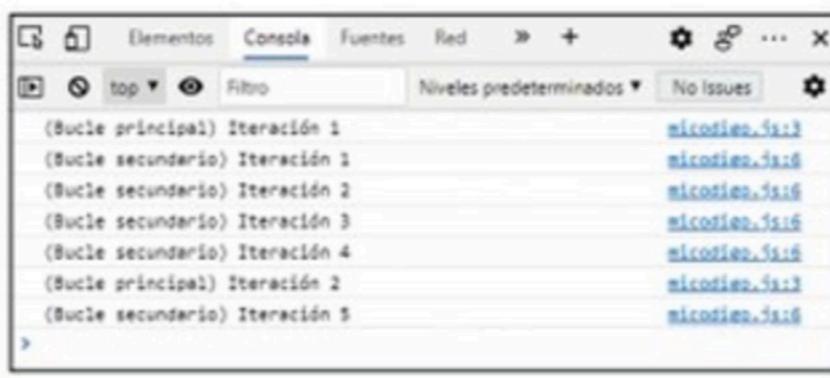


Figura 2.35. Ejecución de una sentencia labeled con break.

En este código hay dos bucles **while** anidados. Como puede verse, el principal se ha etiquetado como **buclePrincipal**. Ninguno de los dos bucles tiene criterio de terminación en su condición principal (**true**). Están diseñados para que se ejecuten indefinidamente, o hasta que otra instrucción los rompa, como en este caso. Lo que se desea es que cuando las variables alcancen ciertos valores se rompa un bucle u otro en función de ciertos intereses. Tal y como se observa en el resultado de la ejecución, hay dos **break** situados dentro del **while** anidado. Llegado el momento, **break** rompe el bucle más próximo que lo envuelve, y **break buclePrincipal** rompe el bucle etiquetado con ese identificador.

Las instrucciones de salto **break**, **continue** y **labeled** son recursos que deben utilizarse en muy contadas ocasiones. El motivo es que abusar de ellas produce algoritmos denominados «spaghetti code» debido a la cantidad de saltos adelante y atrás que es necesario realizar para seguir el flujo de ejecución del programa. Esto es completamente contrario a las normas de estilo más elementales de la programación, donde se busca simplicidad y legibilidad en pro de un software de mantenimiento más sencillo. En muchas empresas está completamente prohibido su uso, salvo casos muy excepcionales.

