UD3. PRINCIPIOS DE PROGRAMACIÓN EN JAVASCRIPT

1. EJECUTAR CÓDIGO JAVASCRIPT

1.1. EJECUCIÓN DE JAVASCRIPT EN EL NAVEGADOR

JavaScript nació para ser un lenguaje interpretado por un navegador. Por ello ejecutar JavaScript desde el código de una página web, sigue siendo considerado como el método principal de ejecución de este lenguaje. Los navegadores interpretan todo el código JavaScript que se encuentre dentro de una etiqueta script. Ejemplo:

```
<!DOCTYPE html>
<html>
<head lang="es">
<meta charset=”UTF-8”>
<title>Prueba JavasCript</title>
</head>

<body>
<script>
        document.write("Hola desde JavaScript");
</script>
</body>
</html>
```

La etiqueta **script** contiene el código JavaScript, que en el ejemplo anterior provoca (gracias al método **document.write**) que se escriba el texto ***Hola desde Javascript*** en el cuerpo de nuestra página web.

La etiqueta script se puede colocar tanto en la cabecera del documento (dentro de **head**) como dentro de la propia etiqueta **body**. Normalmente, se coloca en la cabecera el código para cargar librerías general o código que define funciones y objetos globales, pero que no interacciona con el documento. Se coloca en el cuerpo y justo antes del body, el código JavaScript que se comunica con los objetos del documento.

Ahora bien, el código JavaScript, aun siendo colocado al final de body puede ejecutar su código antes de que se carguen todos los elementos de la página, ya que la carga de una página web se realiza de forma asíncrona. De cómo solucionar este problema hablaremos más adelante.

1.1.1. ESCRITURA POR CONSOLA

Todos los navegadores actuales poseen una ventana de depuración para poder examinar, modificar y testear el código de las aplicaciones web a fin de verificar su eficacia y de poder hacer pruebas y ensayos en él. El lenguaje JavaScript posee un objeto llamado **console** que permite utilizar la consola de depuración desde el propio código. El método **log** se utiliza para escribir en la consola, de modo que esta página web:

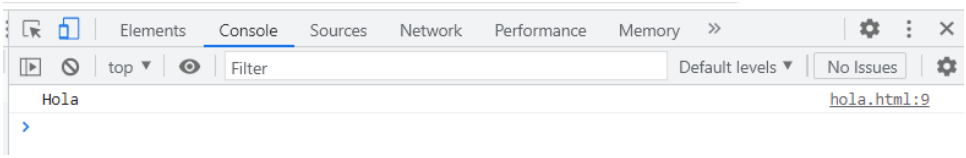
```
<!DOCTYPE html>
<html>
<head lang="es">
<meta charset=”UTF-8”>
```

```
<title>Document</title>
</head>

<body>
<script>
        console.log("Hola”);
</script>
</body>
</html>
```

Si ejecutamos este código en un navegador, no muestra nada en la ventana, pero en el panel de depuración veremos escrito el texto **Hola**.

Podemos ver la consola, dentro del panel de depuración, en prácticamente todos los navegadores pulsando F12 (algunos navegadores usan otras teclas). El panel de depuración sirve para hacer pruebas y detectar fallos en nuestras aplicaciones web. En este panel siempre hay una pestaña llamada **Consola** (**Console**).



1.1.2. CARGAR ARCHIVOS EXTERNOS JAVASCRIPT

Si queremos, podemos escribir el código en otro archivo separado (al que se le debe colocar la extensión **js**) y, desde el código HTML, cargar su código. Para ello, la etiqueta **script** dispone de un atributo llamado **src** al que se le indica la ruta (URL) del archivo que queremos cargar:

```
<script src=”js/acciones.js”></script>
```

Esta etiqueta carga y ejecuta el código JavaScript del archivo **acciones.js** que se encuentra dentro del directorio **js** que estará, a su vez, en el mismo directorio en el que se encuentra la aplicación web.

- ❑ **ACTIVIDAD 1** --- EJECUTAR JAVASCRIPT EN ARCHIVO EXTERNO ---

1.1.3. ATRIBUTOS DE LA ETIQUETA SCRIPT

La etiqueta script, en HTML 4, requería, obligatoriamente, utilizar un atributo llamado type cuyo valor era el lenguaje concreto que se usa en el script (normalmente **type=”JavaScript”**). Actualmente, desde HTML 5, este atributo es opcional porque se da por hecho que lo normal es que el código sea JavaScript.

Cuando se cargan archivos externos, es decir, si se usa el atributo *src*, disponemos de más atributos:

- El atributo **async**[1] que puede tomar los valores **true** o **false**. Con valor true hace que la ejecución del código JavaScript sea asíncrona. Es decir, se carga el código de forma independiente a la carga de elementos de la página.

[1] **<script async>**: el script se descarga de forma asíncrona, es decir, sin detener el análisis HTML, pero una vez descargado, si se detiene para ejecutar el script. Tras la ejecución se reanuda el análisis HTML. Sigue existiendo un bloqueo en el

- El atributo **defer**[2] (con los posibles valores de **true** o **false**) en caso de valer true, indica que la carga del archivo se realizará cuando los componentes de la página web se hayan ya cargados. Es útil (si el navegador entiende este atributo) cuando el código que queremos cargar sirve para manipular el contenido de la página web.
- El atributo **integrity** se usa para poder verificar que el archivo JavaScript que estamos cargando realmente es el que deseamos cargar y no se ha manipulado de forma malévola en el servidor que lo contiene por parte de terceros. Para ello este atributo puede tomar un valor **hash** que el navegador puede usar para verificar el archivo.

  Actualmente, la librería de JavaScript más utilizada es **JQuery** que, hace un tiempo, tuvo el problema de que su código fue modificado con malas intenciones en sus propios servidores de Internet. Por ello, se recomienda cargar de forma remota utilizando este atributo. Por ejemplo:

```
<script src="https://code.jquery.com/jquery-3.5.1.min.js"
integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
crossorigin="anonymous"></script>
```

  La carga de JavaScript desde CDN[3] pedirá al navegador comprobar que al aplicar el algoritmo SHA-256 en el mismo, el código hash resultante es el que se indica. En definitiva, asegura que el archivo cargado no ha sido manipulado por un tercero.

  Ejemplo: Aplicación online que genera el hash de 16 bits de un archivo https://md5file.com/calculator

  Desde el terminal de Ubuntu podemos generar el hash ejecutando:

```
$ cat greeting.js | openssl dgst -sha384 -binary | openssl base64 -A
```

- El atributo **crossorigin** permite establecer atributos relacionados con **CORS** (**Cross-Origin Resource Sharing**). Es una técnica que permite añadir atributos a las peticiones http que, a veces, se necesitan para poder acceder a recursos de servidores remotos. El atributo **crossorigin** de la etiqueta script, permite configurar algunos de esos parámetros. En el caso de enviar credenciales podría ser una cookie, un certificado, una autenticación HTTP Basic…

```
<script crossorigin="anonymous|use-credentials">
```

1.2. EJECUCIÓN DE JAVASCRIPT FUERA DEL NAVEGADOR

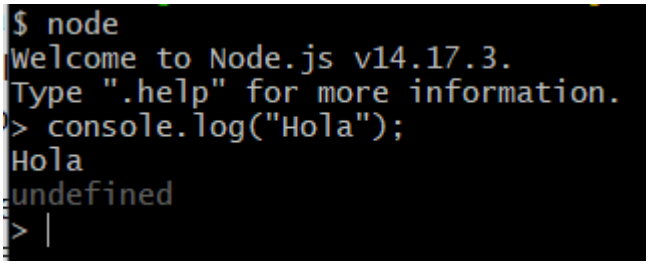
Si deseamos ejecutar JavaScript fuera del navegador, necesitamos la ayuda de **node.js**. Podemos ejecutar directamente código JavaScript de forma rápida si, desde un terminal del sistema, invocamos

renderizado pero menor que con el comportamiento normal. **No se garantiza la ejecución de los scripts asíncronos en el mismo orden en el aparecen en el documento**.

[2] **<script defer>**: el script se descarga de forma asíncrona, en paralelo con el análisis HTML, y además su ejecución es diferida hasta que termine el análisis HTML. No hay bloqueo en el renderizado HTML. La ejecución de todos los scripts diferidos se realiza en el mismo orden en el que aparecen en el documento.

[3] Un CDN es una red de distribución de contenidos (Content Distribution Network) una red pensada para descargar contenidos de forma veloz. (Funcionan como una red repartida geográficamente que permite que accedas al contenido de forma rápida desde cualquier parte del mundo, aunque la página esté en la otra punta.)

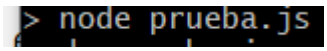
a node y (tras la aparición del símbolo “>”) escribimos el código deseado. Al pulsar Intro se ejecutará ese código. Ejemplo:



Vemos cómo el método **console.log** escribe directamente la palabra **Hola**. En node.js no hay página web y **console.log** muestra directamente en el terminal el resultado de escribir la consola. Tras la escritura de la palabra **Hola**, en la línea siguiente se muestra el texto **undefined** que, lo que hace es escribir el valor que devuelve la expresión (como esta expresión no retorna valor alguno, indica que el valor es indefinido)

Podemos seguir escribiendo código JavaScript, hasta que decidamos salir del entorno pulsando dos veces la combinación de teclas **Ctrl +C** o 1 vez **Ctrl + D.**

También podemos escribir código JavaScript en un archivo (es recomendable que tenga extensión **js)** y después ejecutar el código de ese archivo utilizando node. Supongamos que hemos grabado un archivo que se llama **prueba.js**, entonces desde el terminal nos debemos situar en el directorio en el que está el archivo y ejecutar este comando:



Veremos en el terminal el texto que el programa escriba por consola.

- ❑ **ACTIVIDAD 2** – EJECUTAR JAVASCRIPT DESDE NODE --

2. FUNDAMENTOS BÁSICOS

JavaScript es un lenguaje de programación cuya sintaxis y forma de escribir sentencias recuerda a la mayoría de los lenguajes que han tenido éxito en estas últimas décadas. Esto significa que, para personas que sepan programar en algún lenguaje de programación, el aprendizaje de las bases fundamentales, resulta muy sencillo.

2.1. CONCEPTOS FUNDAMENTALES

2.1.1. ¿JAVASCRIPT ES UN LENGUAJE INTERPRETADO?

Desde hace muchos años se ha hecho una división de los distintos lenguajes de programación entre los que son **compilados** y los que son **interpretados.**

Antes de la década de los 90 del sigo XX, la mayoría de lenguajes que florecían eran compilados. La razón es que el código de un lenguaje compilado se convierte a código máquina, analizando todas las líneas del mismo antes de empezar a convertir a binario el código. De esta forma, se consigue un código

más eficiente. Si hay un error, por ejemplo, en la línea de código número 200, en el proceso de compilación se nos indicará el error de esa línea y no veremos el resultado que producen las primeras 199 líneas (que son correctas) si no arreglamos el problema de la línea 200.

En los lenguajes interpretados se realiza la conversión a código máquina línea a línea. Usando el mismo ejemplo, si tenemos un error en la línea 200 veríamos con normalidad el resultado de las primeras 199 líneas. Sería al intentar interpretar la línea 200 cuando se notificaría del error.

Sin embargo, el código interpretado ofrece ventajas cuando se debe ejecutar en un entorno donde la cargar del programa es lenta, como ocurre en Internet (aunque cada vez menos). El hecho de que sea interpretado permite ir ejecutando las primeras líneas que se carguen antes incluso de que se carguen las siguientes, lo cual es mucho más eficiente en el caso de código incrustado en el lado del cliente de las aplicaciones web.

Tradicionalmente, se considera a JavaScript un lenguaje interpretado. Pero la realidad es más compleja. Inicialmente JavaScript solo era apto para realizar acciones no muy complejas, hoy en día sus capacidades no le diferencian de la mayoría de los lenguajes existentes, y eso ha implicado que los navegadores precompilen el código JavaScript para que sea más veloz su rendimiento.

Actualmente, JavaScript utiliza un compilador **JIT (Just In Time)** gracias a una máquina virtual de JavaScript (**JVM**) que proporcionan los navegadores. Diremos que JavaScript es un lenguaje con una compilación semejante a la del lenguaje Java. El código de JavaScript se compila al vuelo (ya que se debe ejecutar lo más rápido posible) para que se pueda servir lo más rápido posible, pero habiendo optimizado el código.

No se consigue la rapidez de lenguajes totalmente compilados como el lenguaje C, pero tampoco adolece de la falta de eficiencia del lenguaje JavaScript original. Al final, la velocidad de este proceso de conversión de código depende de los trucos y habilidades que proporcionan los motores de ejecución de JavaScript (programas que ejecutan código JavaScript). El famoso V8 del navegador Chrome de Google, consiguió acelerar la ejecución de código JavaScript de forma notable (otros motores son JavaScriptCore de Apple, SpiderMonkey de Mozilla o Chakra de Microsoft).

En todo caso, sí conviene pensar que JavaScript tiene mucho todavía de lenguaje interpretado, y que no todos los errores que cometamos se marcarán al comenzar a ejecutar nuestro código.

2.1.2. SENTENCIA O INSTRUCCIÓN

Como todos los lenguajes de programación, el código se divide en sentencias o instrucciones. Se trata de la estructura mínima del programa. Una sentencia es una estructura básica de código que es capaz de realizar una acción simple. Podemos entender un programa como una sucesión de sentencias o instrucciones.

En JavaScript las instrucciones simples se pueden finalizar con el símbolo de punto y coma (;). Sin embargo, no es obligatorio y podemos escribir instrucciones sin indicar el final. Nuestra recomendación es finalizar siempre con punto y coma, ya que esto posibilita un mejor aprendizaje del lenguaje si entendemos cuándo debemos indicar el punto y coma.

Un ejemplo de sentencia es:

```
var nombre=”Gema”;
```

Lo que hace es asignar el texto **Gema** a una variable llamada **nombre**.

2.1.3. IDENTIFICADORES Y PALABRAS RESERVADAS

La palabra **var** del código anterior es lo que se conoce como **palabra reservada**. Un término especial que, dentro del lenguaje, tiene un significado especial.

En cualquier lenguaje de programación se puede dar nombre a elementos que el programador necesita en su código. Por ejemplo, en el código anterior **nombre** es como hemos decidido denominar a la variable que almacena la palabra **Gema**, que se entrecomilla porque es un texto. No podríamos haber llamado a esa variable **var**, como tampoco **if** o **try**, porque todas ellas son palabras reservadas del lenguaje.

Es decir, las palabras reservadas son términos que ofrece el lenguaje para poder programar. Mientras que los nombres que damos a variables y otros elementos del lenguaje se conocen como **identificadores**, ya que su labor es precisamente identificar de forma inequívoca a un elemento del lenguaje.

Los identificadores deben de cumplir las siguientes reglas:

- Deben comenzar por una letra, un guión bajo (_) o el símbolo dólar ($)
- Después se pueden usar letras, números o el guión bajo.
- Se pueden usar letras Unicode, aunque estén fuera del código ASCII, como á o é.
- Se distingue entre mayúsculas y minúsculas. **SalarioFinal** es un identificador distinto de **SalarioFinal**

2.1.4. REGLAS FUNDAMENTALES

- **Case Sensitive**. JavaScript es un lenguaje sensible a mayúsculas. Es decir, se distingue entre mayúsculas. Es decir, se distingue entre mayúsculas y minúsculas. No es lo mismo var que VAR, por eso este código produce un error:

```
VAR x=7; // ¡Error!
```

- Los **comentarios de una línea** comienzan con los símbolos //. Un comentario es un texto explicativo que no se tendrá en cuenta al interpretar el código, y por ello, se usa para documentar código. De esa forma podemos explicar mejor lo que hacemos en el código especialmente si trabajamos en equipo, pero también incluso para nosotros mismos ya que, con el paso del tiempo, se nos olvidará el funcionamiento de muchas de las instrucciones que escribamos.

```
var tam=nombre.length(); //tam recoge el tamaño del nombre
```

- Los **comentarios de varias líneas** comienzan con los símbolos /* y terminan con */

```
/* Comienzo de la función principal
De proceso de datos */
```

- Se usan cuando necesitamos escribir mucho texto explicativo. Aunque están pensadas para comentarios largos, se pueden utilizar en pequeños bloques de una línea:

```
var n = /*Ejemplo*/ 17;
```

La variable identificada como n valdrá **17.**

- Se pueden agrupar sentencias de **bloques** de sentencias. Los bloques son sentencias contenidas entre llaves (símbolos { y }). Normalmente los bloques forman parte de sentencias complejas. Ejemplo:

```
var x=9;
if (x==9){

        x++;
        console.log(x);
}
```

En este ejemplo, la sentencia if contiene un bloque de sentencias, las cuales se ejecutan si se cumple la condición x==9. En este caso, si probamos el código veremos que, por consola, se escribe el número 10.

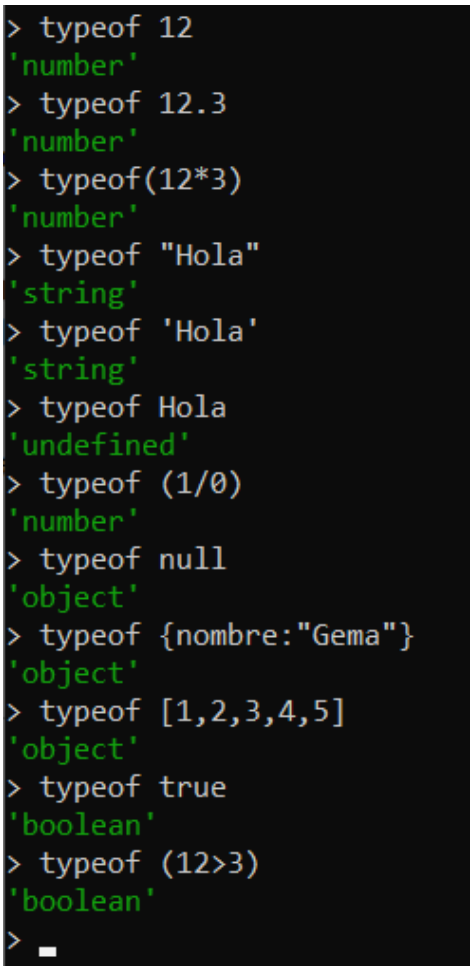
3. VARIABLES Y TIPOS DE DATOS

3.1. VALORES

Los programas en JavaScript utilizan valores. Los valores son datos que el programa necesita almacenar, manipular o mostrar.

En JavaScript hay 3 tipos de valores básicos: **Textos** (llamados **Strings**), **Números** y **Valores booleanos**. A estos 3 tipos hay que añadir el tipo complejo **Objeto** y los valores **indefinido** y **nulo**.

Existe un operador llamado **typeof** que permite conocer el tipo JavaScript de un valor concreto. Vamos a ver algunos ejemplos de uso de estos valores. Para ello vamos a invocar a **node** desde la línea de comandos, y pediremos que nos escriba los tipos correspondientes a varios valores:



- El operador **typeof** valora la expresión que escribamos a su derecha. Si es compleja, se debe indicar entre paréntesis para asegurar que se evalúa el conjunto completo. La respuesta de este operador es un texto que indica el tipo de datos al que pertenece la expresión.
- Todos los números tienen el tipo (**number**) sin importar si son decimales o no. Incluso se devuelve el tipo **number** para la expresión **1/0** que realmente no se puede evaluar. Sin embargo, se considera que uno dividido entre 0 produce infinito, y en JavaScript, el infinito es un número válido.
- Los textos se entrecomillan. Cualquier expresión entrecomillada (comillas simples o dobles) se entiende que es un texto (**string**).
- El tipo **undefined** se emplea cuando JavaScript no puede evaluar una expresión. La expresión **typeof hola** provoca un resultado indefinido porque se busca la variable hola (no está entrecomillada) y, como no existe, se decide que es una expresión indefinida.
- Los tipos complejos (**object**) se utilizan mucho. Incluso el valor **null** se considera un objeto.

3.2. DECLARACIÓN DE VARIABLES

En los lenguajes de programación, los valores se almacenan en variables. Las variables son uno de los elementos fundamentales en cualquier lenguaje de programación.

Cada variable tiene un identificador. A ese identificador se le asocian los valores que deseemos. Lo interesante es que podemos operar con las variables para realizar los cálculos que deseemos.

En JavaScript las variables deben de ser declaradas antes de poderse usar. Declarar una variable es asignarla un identificador. Tradicionalmente la palabra reservada **var** es la que se utiliza para declarar una variable:

```
var x;
```

De esta forma estamos avisando de que se podrá utilizar, a partir de ese momento, en el código una variable que se llama x.

Es muy habitual asignar un valor al declarar una variable:

```
var x=9;
```

Con esto declaramos la existencia de una variable que se llama **x**. Además, estamos indicando que, inicialmente, vale 9.

Las variables solo se declaran una vez. Después la variable se usa con normalidad:

```
var x = 9;
x = 25;
console.log(x); //escribe 25
```

3.2.1. DIFERNCIA ENTER LET Y VAR

Desde la versión 6 del estándar **ECMAScript** (**ES2015** o **ES6**), se permite el uso de las palabras reservadas **let** y **const** para declarar variables (además de usar **var**).

let se utiliza para declarar variables de forma más local que **var**. Para entender la diferencia veamos este código:

```
{
    let x=9;
};
console.log(x);
```

Si ejecutamos este código, aparecerá el error: "ReferenceError: x is not defined.

La razón es que la variable **x** se ha definido dentro de un bloque mediante la palabra clave **let** y no se puede utilizar fuera. Digamos que solo existe esa variable dentro del ámbito del bloque en el que se declaró.

Sin embargo, este código funciona sin errores:

```
{
   var x= 9;
}
Console.log(x);
```

Ahora aparece el número 9 como resultado de ejecutar el código.

Por lo tanto, la diferencia es que **let** restringe el uso de la variable al bloque concreto en el que se encuentra. En el caso de **var**, el ámbito de aplicación es mayor, la variable sobrevive fuera del bloque.

Realmente las variables declaradas con **var** también tienen restricciones en cuanto a su ámbito de uso. Vamos a ver este código:

```
function f(){
   var x=9;
}
console.log(x);
```

Al ejecutar el código, volvemos a producir un error de variable indefinida, puesto que las funciones también utilizan bloques de código. La diferencia, ahora, es que esos bloques son más restrictivos: incluso las variables definidas por **var** no pueden utilizarse fuera del bloque definido por una función.

Siendo más formales, se dice que el ámbito de las variables declaradas con **let** es el bloque, y el ámbito de uso de las variables declaradas con **var** es el contexto de ejecución en el que se encuentra la palabra **var**. Este último ámbito puede ser **global**, si se declara fuera de la función, y abarca todo el archivo, o puede ser relativo a una función si se declara dentro de ella.

Este otro código nos ayuda a reflexionar sobre la idea:

```
var x=7; //x se declara en ámbito global
function f(){
    var x=9; //x se declara en el ámbito de la función
}
console.log(x);
```

El resultado de ese código es la escritura por pantalla del número 7. Realmente hay 2 variables llamadas **x**, la primera tiene ámbito global y la segunda se restringe a la función. Por eso, como **console.log(x)** está fuera de la función, se usa la x global, que vale 7.

3.2.2. DECLARACIÓN CON CONST

Este término se usa para definir constantes. Funciona bajo las mismas condiciones de ámbito que **let**. La diferencia es que una variable declarada con **const** no puede modificar su valor.

```
const PI=3.14592;
PI=9;
```

La ejecución de este código produce un error:

TypeError: Assignment to constant variable

No podemos modificar el valor de una variable declarada mediante la palabra **const**.

Aunque se pueden usar tanto minúsculas como mayúsculas para declarar constantes puras, es recomendable usar mayúsculas, ya que así las distinguimos más fácilmente de las variables.

3.3. TIPOS DE DATOS PRIMITIVOS

3.3.1. NÚMEROS

A diferencia de otros lenguajes (como **C**, **C++** o **Java**); JavaScript utiliza un solo tipo de datos para los números, sin importar si el número tiene decimales o no.

Los números se escriben tal cual, sin comillas ni símbolos extra. Si tienen decimales se usa el punto como separador decimal.

```
let entero=1980;
let decimal=0.21;
```

El hecho de no diferenciar enteros de números decimales, hace que todos los números ocupen realmente el mismo espacio en memoria: **64 bits.**

Otras formas de indicar números

JavaScript admite formatos especiales numéricos. Por ejemplo, admite la notación científica:

```
const AVOGADRO=6.022e+23;
```

Lo cual significa: $6{,}022.10^{23}$

La notación científica es ideal para números muy grandes o muy pequeños.

Por otro lado, no solo se aceptan números decimales, también se aceptan números hexadecimales si se les antepone el prefijo **0x**.

```
let hexa=0xAB12;
console.log(hex);
```

El resultado es **43794**, resultado de convertir a decimal el número hexadecimal **AB12**.

También es posible usar números octales. Estos usan el cero como prefijo seguido de la letra o minúscula:

```
let octal=0o27652;
console.log(octal);
```

Ahora aparece el número 12202, resultado de convertir a decimal el octal 27652.

Por si fuera poco, podemos también indicar números en binario mediante el prefijo **0b**.

```
let binario=0b10111011;
console.log(binario);
```

El número binario se convertirá al decimal correspondiente (187).

Números especiales

JavaScript permite utilizar el número infinito (**Infinity**). Ejemplo:

```
var x=1/0;
console.log(x);
```

Por pantalla aparece **Infinity**.

```
var y=Infinitiy;
console.log(y+1);
```

Este código también produce la escritura de **Infinity**.

Hay otro término especial.

```
var x="hola" * 3;
console.log(x);
```

Aparece el código **NaN**(**Not a Number**), que se provoca cuando una expresión intenta operar de forma numérica con un valor que no es número. Este término facilita la depuración de errores en JavaScript.

Las indeterminaciones matemáticas también provocan este valor. Por ejemplo, la expresión 0/0 o **Infinity – Infinity**. Estas habilidades matemáticas de JavaScript permiten resolver de forma muy notable problemas complejos sobre números.

3.3.2. STRINGS

Delimitar strings, cadenas de caracteres o textos.

Se pueden utilizar tanto las comillas simples, como las dobles.

```
texto=”Mi apellido es Morant”;
texto=’Ricardo no para de hablar’;
```

Esto permite que dentro del texto puedan aparecer comillas dobles o simples:

```
frase= “Mi apellido es O’Moran”;
frase2=’Antelo vino y me dijo “Hola”’;
```

Desde la versión 6 se permiten también las comillas invertidas (**backticks**).

```
Texto=`Hola a todos`;
```

Uso de plantillas de string

La tercera forma de delimitar textos, con el uso de comillas invertidas, permite, de forma muy sencilla, colocar expresiones dentro de comillas.

```
var nombre=”Gema”;
console.log(“Me llamo nombre”);
```

Por pantalla, la ejecución del código escribe **Me llamo nombre**. Pero, es posible que pretendiéramos escribir **Me llamo Gema**. Pero, entre comillas el texto se toma como literal. Hay una solución gracias al operador de concatenación (+).

```
var nombre=”Gema”;
console.log(“Me llamo “ + nombre);
```

Ahora el resultado muestra **Me llamo Gema**

Aunque el operador de concatenación es muy interesante, si el texto a escribir es complejo, es pesado tener que encadenar expresiones con el operador +.

Las plantillas de string aportan otra solución. Se llaman plantillas de string (**String templates**) a los textos delimitados con comillas invertidas. La cuestión es que se admite incluir el signo **$** dentro del texto delimitado y después, entre llaves, indicar una expresión JavaScript. El texto dentro de llaves no se tomará como texto literal, se interpretará como si estuviera fuera de las comillas. Por ejemplo:

```
var nombre=”Gema”;
console.log(`Me llamo ${nombre}`);
```

Nuevamente vemos **Me llamo Gema**.

Podemos incluso indicar expresiones más complejas:

```
console.log(`Cada día hay ${24*60*60} segundos`);
```

Escribe: Cada día hay 86400 segundos

Secuencias de escape

Hay caracteres que no se pueden escribir y que, sin embargo, son muy necesarios. Es el caso del salto de párrafo (relacionado con la tecla Intro) o el tabulador. Se consiguen escribir esos caracteres usando un código especial que comienza con la barra inclinada a la izquierda (backslash) seguida del código concreto. Por ejemplo.

```
let texto="Una línea\nOtra línea";
```

Por consola aparece el texto **Una línea** en la primera línea y debajo el texto **Otra línea**.

Algunas de las secuencias más utilizadas son:

SECUENCIA	SIGNIFICADO
\n	Salto de línea
\t	Tabulador
\r	Retorno de carro
\f	Salto de página
\v	Tabulador vertical
\"	Comillas dobles
\'	Comillas simples
\b	Retroceso
\\	El propio carácter backslash

Ejemplo de uso de comillas usando secuencias de escape:

```
let texto="Paseando por la calle O'Donell dijo \"¡Qué calor!\"";
```

Es necesario usar las comillas dobles del texto "¡Qué calor!" como secuencia de escape, para distinguirlas de las comillas dobles que sirven para delimitar el texto.

Es posible también usar secuencias de escape para poder almacenar en strings caracteres Unicode. Con esa finalidad se utiliza el código **\u** seguido de 4 cifras hexadecimales. Ejemplo:

```
console.log( "\u2A1D");  //Escribe el símbolo ⋈
```

En el ejemplo anterior hemos conseguido escribir el símbolo correspondiente a la operación **JOIN** de las bases de datos relacionales: ⋈

La tabla Unicode de caracteres tiene incluso más símbolos que los que pueden mostrarse con 4 cifras hexadecimales (16 bits), pueden usar incluso el doble de bits. Para este tipo de símbolos se usa el formato: **\u{código}** siendo el código el número hexadecimal de hasta 8 cifras (32 bits). Ejemplo:

```
console.log("\u{1F48B}");
```

Escribe el símbolo 💋, el clásico emoticono de marca de labios.

3.3.3. BOOLEANOS

Los valores booleanos solo pueden tomar los valores **true** o **false**. Estos valores son palabras reservadas que se pueden asignar directamente:

```
let b=true;
```

Sin embargo, suele ser más habitual que esos valores se produzcan al evaluar una expresión lógica:

```
let x=9;

let y=10;

let b=(x>y);  //b vale false porque x no es mayor que y
```

Incluso podemos decidir que toda expresión y valor en JavaScript se asocia con valores verdaderos o falsos. Para poder comprobar que esto es cierto, podemos usar la función **Boolean** que se escribe el valor booleano equivalente para cualquier valor:

Console.log(Boolean(**true**));	//Escribe true
Console.log(Boolean(1));	//Escribe true
Console.log(Boolean(0));	//Escribe false
Console.log(Boolean("Hola"));	//Escribe true
Console.log(Boolean(""));	//Escribe false
Console.log(Boolean(NaN));	//Escribe false
Console.log(Boolean(undefined));	//Escribe false
Console.log(Boolean(Infinity);	//Escribe true
Console.log(Boolean(null));	//Escribe false

En realidad, solo los textos vacíos (comillas sin contenido), el valor cero, el valor false, el valor **NaN**, el valor **undefined** el valor **null**, se consideran falsos.

Este hecho se puede utilizar para poder evaluar de forma sencilla valores en instrucciones de control como, por ejemplo, la sentencia **if**.

4. OPERADORES

4.1. ARITMÉTICOS

Los operadores aritméticos permiten realizar operaciones matemáticas sobre los números. Son los siguientes:

OPERADOR	SIGNIFICADO	EJEMPLO
+	Suma	console.log(5 + 4.5); //Escribe 9.5
-	Resta	console.log(5 - 4.5); //Escribe 0.5
*	Multiplicación	console.log(5 * 4.5); //Escribe 22.5
/	División	console.log(7 / 3); //Escribe 2.333333
%	Resto	console.log(7 % 3); //Escribe 1
**	Exponente	console.log(3 ** 2); //Escribe 9

El exponente (**) se añadió en la versión 2017 del estándar.

4.2. RELACIONALES

Los operadores relacionales permiten comparar expresiones. El resultado de la operación es siempre un valor booleano. Son:

OPERADOR	SIGNIFICADO	EJEMPLO

>	Mayor	console.log(4 >5); //Escribe true
		console.log(5 >5); //Escribe false

<	Menor	console.log(5 < 4); //Escribe false
		console.log(5 < 5); //Escribe true

>=	Mayor o igual	console.log(5 >= 4); //Escribe true
		console.log(5 >=5); //Escribe true

<=	Menor o igual	console.log(5 <= 4); //Escribe false
		console.log(5 <= 5); //Escribe true

==	Igual	console.log(5 == 4); //Escribe false
		console.log(5 == 5); //Escribe true

!=	Distinto	console.log(5 != 4); //Escribe true
		console.log(5 != 5); //Escribe false

Un error muy habitual es confundir el operador de asignación (=) con el operador de valoración de igualdad (==). El primero permite asignar un valor a una variable, el segundo compara y devuelve verdadero o falso en función de si se cumple o no la igualdad. Veamos un ejemplo:

```
console.log(5=4);
```

Esa expresión provoca un error de **ReferenceError: Invalid left-hand side in assignment.**

En cambio:

```
concole.log(5==4);
```

Mostrará el valor **false** porque 5 no es igual a 4.

Podemos comparar strings:

```
console.log("saco">"saca");
```

Escribirá true, porque, en orden alfabético, **saco** es mayor que **saca**. Sin embargo, esta expresión:

```
console.log("Saco">"saca");
```

Escribe, false.

Lo mismo ocurre con algunos símbolos:

```
console.log("ñ"<"o"); //Escribe false
```

A la **eñe** se le asigna un código mucho mayor que el que se le asigna a la letra **o**, lo que invalida estos operadores para ser utilizados con textos en español. Como veremos más adelante la comparación correcta debe utilizar el método **localCompare**.

4.2.1. COMPARACIONES ESTRICTAS

JavaScript posee un operador de comparación estricta que utiliza 3 signos de igual (===). La razón procede de la capacidad de JavaScript de comparar datos de diferente tipo:

```
console.log("2"==2);
```

Esa expresión escribe **true**, porque, aunque "2" es un texto, JavaScript puede comparar de forma numérica el contenido de ese texto. Sin embargo:

```
console.log("true"==true);  //Escribe false
```

No funciona con todo tipo de datos, el texto **true** no se puede convertir a su equivalente booleano.

```
console.log(1==true);  //Escribe true
```

Estamos comparando un número con un valor booleano, el resultado es true porque se considera que 1 es perfectamente convertible al valor true. Algo que ocurre con el cero también.

```
console.log(0==false); //Escribe true
```

Con otros números:

```
Console.log(2==false); //Escribe false
Console.log(2==true);  // Escribe false
```

Esto se debe a fin de compatibilizar el código con la forma de usar true y false con 1 y 0 en algunos lenguajes (como C). Incluso:

```
console.log("1"==true);  // Escribe true
console.log(undefined==null);  //Escribe true
```

Pero,

```
 console.log(undefined===null); //Escribe false
Console.log(undefined===false; //Escribe false
```

En JavaScript es muy habitual que recojamos números que el usuario escribe y se reciben como strings. La siguiente comparación devuelve verdadero:

```
console.log("2"==2);  // Escribe true
```

Si necesitamos comparar de forma más estricta:

```
console.log("2"===2);  //Escribe false
console.log(1+1===2);  //Escribe true
```

Para la comparación de la negación:

```
console.log("2"!=2);  //Escribe false
console.log("2"!==2); //Escribe true
```

La primera sentencia escribe false porque considera iguales 2 y "2". Sin embargo, el operador de diferencia estricta escribe true porque entiende que el 2 no es igual como texto que como número.

4.3. OPERADOR DE ENCADENAMIENTO

El signo (+) en JavaScript tiene 2 posibilidades. Suma de números y concatenación. Ejemplo:

```
let hoy="Hoy es ";
let dia="Viernes";
console.log(hoy + dia);
```

Por pantalla aparece **Hoy es viernes**.

4.4. OPERADORES LÓGICOS

Las expresiones lógicas permiten valorar condiciones para tomar decisiones en nuestro programa. Los operadores lógicos permiten utilizar expresiones lógicas complejas. Estos operadores son:

OPERADOR	SIGNIFICADO
!	NOT
&&	AND
||	OR

El operador AND (&&) permite comparar 2 expresiones lógicas. El resultado será verdadero si las 2 expresiones que se comparan lo son.

```
let x=9;
let y=10;
console.log(x==9 && y==10); //true
let x=9;
let y=12;
console.log(x=9 && y==10); //false
```

El operador OR (||) compara 2 expresiones devolviendo true si cualquiera de las 2 expresiones (o las 2) son verdaderas.

```
let x=9;
let y=12;
console.log(x=9 || y==10); //true
```

Finalmente, el operador NOT(!) niega la expresión que tenga a su derecha: si es falsa devuelve true y si es verdadera, devuelve false:

```
let x=9;
let y=12;
console.log(!(x==9) || y==10)); //false
```

Hay que tener cuidado con este operador, ya que tiene más prioridad que los operadores AND y OR, por lo que si en el código retiramos los paréntesis:

```
let x=11;
let y=12;
console.log(!x==9 || y==12); //true
```

La expresión, en este caso, **x==9** devolvería false, pero como está matizada con el operador **NOT**, devuelve **true**, luego se valora la expresión **y==12** (que es true) y luego se une ambas con el operador **OR**, que resulta **true**. Todo cambiaría con un paréntesis:

```
let x=11;
let y=12;
console.log(!(x==9 || y==12)); //false
```

Ahora se evalúa primero **x==9** que es false, luego **y==12** que es true. Se unen ambas expresiones con **OR** que devolvería true. Pero al final, el **NOT** provocaría el resultado false.

Un hecho interesante del operador AND es que la segunda expresión solo se evalúa si la primera es verdadera, ya que si la primera es falsa ya sabemos que el resultado será falso. Es decir:

```
let edad=17;

let conduce=(edad>=18 && carnet==true);
```

La variable conduce será falsa porque la **edad** es de 17. Es más, a lo mejor ni hemos declarado la variable **carnet**, lo que, provocaría un error por indefinición de variable. Pero la realidad es que no se va a evaluar la segunda expresión porque la edad no es mayor o igual a 1.

4.5. OPERADORES DE BIT

La realidad de la computación es que solo maneja números binarios. Por lo que, cualquier número, internamente, se almacena en forma binaria JavaScript aporta operadores para trabajar a nivel de BIT.

OPERADOR	SIGNIFICADO
~	NOT
&	AND
|	OR
^	XOR
>>	Desplazamiento derecho
<<	Desplazamiento izquierdo

Operador NOT:

```
console.log(~25); //Escribe -26
```

En binario (con 32 bits), el 25 se escribe: 00000000000000000000000000011001

El operador NOT invierte los números: 11111111111111111111111111100110 y eso, en decimal, se interpreta como -26 (ya que se usa complemento a 2 como representación binaria).

Por otro lado:

```
console.log(26 & 17); //Escribe 16
```

Veamos por qué se escribe 16

```
26 binario:       00000000000000000000000000011010
17 binario:       00000000000000000000000000010001
                  --------------------------------
Resultado AND:    00000000000000000000000000010000 (16 en decimal)
```

Sin embargo,

```
console.log(26 | 17); // Escribe 27
```

```
 26 binario:      00000000000000000000000000011010
17 binario:       00000000000000000000000000010001
                  --------------------------------
Resultado OR:     00000000000000000000000000011011 (27 en decimal)
```

En el caso de XOR:

```
console.log(26 ^ 17); //Escribe 11
```

```
26 binario:       00000000000000000000000000011010
17 binario:       00000000000000000000000000010001
                  --------------------------------
Resultado XOR:    00000000000000000000000000001011 (11 en decimal)
```

En la actualidad los operadores de desplazamiento se utilizan menos. El operador << añade el número de bits con valor cero a la derecha que se indiquen. Así:

```
console.log(9<<3); //Escribe 72
```

```
9 binario:    00000000000000000000000000001001
              --------------------------------
9<<3:         00000000000000000000000001001000 (72 en decimal)
```

Se han añadido 3 ceros por la derecha y eso es como multiplicar por 3 veces por 2 al número

El desplazamiento derecho:

```
cconsole.log(9>>3); //Escribe 1
```

Ya que:

9 binario: 00000000000000000000000000001001

9>>3: 00000000000000000000000000001001 (1 en decimal)

En la práctica el resultado es equivalente a dividir 3 veces por 2.

En la actualidad los operadores de desplazamiento están en desuso.

4.6. OPERADORES DE ASIGNACIÓN

4.6.1. INCREMENTOS Y DECREMENTOS

En JavaScript es muy habitual tener que realizar la siguiente operación:

```
x=x+1;
```

Equivalente a:

```
x++;
```

Ejemplo de código:

```
let x=8;
x++;
console.log(x); //Escribe 9
```

El operador ++ se llama incremento. Disponemos del operador -- para el decremento.

```
let x=8;
x--;
console.log(x); //Escribe 7
```

Estos operadores se pueden colocar tanto detrás como delante de la variable (x++, ++x).

```
let valor=50, incremento, decremento;
//Ejemplo de postincremento, primero se asigna y luego se incrementa la variable
incremento=valor++; //incremento vale 50 y valor vale 51
decremento=valor --; //decremento vale 51, valor vale 50
valor=50;
//Ejemplo de preincremento, primero se incrementa y luego se asigna la variable
incremento=++valor; //incremento vale 51 y valor también
decremento= --valor; //decremento y valor valen 50
```

En el ejemplo se puede observar la diferencia. En el caso de **postincremento (x++)** o **postdecremento (x--)** si es parte de otra expresión el incremento o decremento se hace después. Es decir, en la expresión **y=x++** la variable y tomaría el valor que tenga **x** (sin incrementar) y luego se incrementaría **x**, pero la variable "**y**", ya no se modifica.

En el caso de los **preincrementos** (**++x**) o **predecrementos** (**–x**) siempre se realiza primero la operación de incrementar o decrementar.

4.6.2. OTROS OPERADORES DE ASIGNACIÓN

Hay otros operadores que permiten abreviar las operaciones de asignación:

```
var x=18;
x+=9; //x vale ahora 27
```

La expresión **x+=9** sirve para abreviar la expresión **x=x+9**. Es un operador de asignación y suma. La lista completa de operadores de asignación es:

OPERADOR	SIGNIFICADO	EJEMPLO	EQUIVALENTE A...
+=	Suma y asignación	x+=5	x=x+5
-=	Resta y asignación	x-5=5	x=x-5
*=	Multiplicación y asignación	x*=5	x=x*5
/*	División y asignación	x/=5	x=x/5
%=	Resto y asignación	x%=5	x=x%5
**=	Exponente y asignación	x**=2	x=x**2
&=	AND de bit y asignación	x&=65	x=x&65
|=	OR de bit y asignación	x|=65	x=x|65
^=	XOR de bit y asignación	x^=65	x=x^65
>>=	Desplazamiento derecho y asignación	x>>=3	x=x>>3
<<=	Desplazamiento izquierdo y asignación	x<<=3	x=x<<3

4.6.3. OPERADOR CONDICIONAL

Se trata de un operador capaz de evaluar una expresión booleana, y si ésta es verdadera dar como resultado un valor y si es falsa dar otro. A este operador se le conoce también como ***inline if*** porque su labor se asemeja mucho a la que realiza la estructura **if**.

De hecho, este operador se suele utilizar cuando deseamos hacer evaluaciones rápidas que normalmente requerirían de una instrucción **if**. La sintaxis es:

```
codición ? valorSiVerdadera : valorSiFalsa
```

Ejemplo:

```
let tipo = (numero%2==0) ? “par” : “impar”;
```

5. CONVERSIÓN DE TIPOS

5.1. CONVERSIÓN AUTOMÁTICA

JavaScript no es un lenguaje muy **tipado**. Lo que significa que, a diferencia de lenguajes más fuertemente tipados como **Java**, no se producen errores con facilidad al mezclar tipos diferentes de datos. El lenguaje JavaScript se basa en buscar la lógica más razonable de cada expresión.

Por ejemplo:

```
console.log('2' * 3);
```

Esta expresión produciría un error en muchos lenguajes de programación al intentar multiplicar un número por un string. Sin embargo, JavaScript realiza conversiones automáticas de tipos de datos, siempre que sea posible. Gracias a ello, el resultado de esa operación es 6.

Pero observemos este código:

```
console.log('2' + 3);
```

Escribirá 23. Ahora tenemos un operador capaz de sumar, cuando opera con números, y de encadenar, cuando opera con strings. En este caso, la conversión automática opera con el número 3, que se convertirá en un string para poder aplicar el encadenamiento. El resultado sería el mismo en este caso:

```
console.log('Hola' * 3);
```

No se produce error, pero se escribe el valor NaN porque no se puede conseguir un valor numérico de esa expresión. Pero vemos claramente que JavaScript se esfuerza por conseguir un resultado coherente sin provocar un error. Y por ello, las siguientes expresiones no provocan error:

```
console.log(null * 3); //Escribe 0
console.log(true * 3); //Escribe 3 (true se asocia con el número 1)
console.log(false * 3); //Escribe 0 (false se asocia con el número 0)
console.log(undefined * 3); //Escribe  NaN
```

5.2. FORZAR CONVERSIONES

Algunas veces queremos forzar la conversión de los números. En este ejemplo:

```
let x='2';
let y='3';
console.log(x + y); // Escribe 23
```

¿Y si quisiéramos sumar de forma numérica?. Para esto disponemos de la función **Number**.

```
let x='2';
let y='3';
```

```
console.log(Number(x) + Number(y)); // Ahora escribe 5
```

Pero no todo se puede pasar a número.

```
console.log(Number("Hola")); // Escribe NaN
```

Disponemos de una función para convertir a strings. Se trata de **String**.

```
let x=2;
let x=3;
console.log(String(x) + String(y)); //Escribe 23
```

Hay una función muy potente llamada **parseInt** para convertir a números enteros. En este sentido es como **Number**, pero con capacidad solo de convertir a números no decimales. Sin embargo, permite cambiar de base y así pasar, por ejemplo, de números binarios a decimales:

```
console.log(parseInt("101101",2));
```

Tras el número a convertir (101101) se coloca una coma y el número 2, indicando que el texto original contiene un número en base 2.

parseInt además tiene la capacidad de extraer el número con el que comienza un texto (el método **Number** no tiene esa capacidad):

```
console.log(parseInt("22veces", 10)); // Escribe 22 en lugar de NaN que haría la función Number
```

Cuando el número está en base decimal, no hace falta explicar el número 10.

```
console.log(parseInt("22veces")); // Escribe 22
```

Pero hay una función llamada **parseFloat** que sí tiene esta capacidad:

```
console.log(parseFloat("22.5veces")); //Escribe 22.5
```

5.3. FUNCIÓN ISNAN

Hay problemas en JavaScript para determinar si una expresión no es numérica. Aunque el valor **NaN** es un valor válido en JavaScript, y sabemos que, por ejemplo, **"Hola"*4** produce un resultado no numérico. La comparación **("Hola"*4)==NaN** produce un resultado falso.

Por ello se utiliza la función llamada **isNaN**. Esta función recibe una expresión y devuelve verdadero si la expresión es no numérica.

```
console.log(isNaN(NaN)); // Escribe true
console.log(isNaN("Hola"*5)); // Escribe true
console.log(isNaN("3"*5)); // Escribe false, "3"*5 retorna 15, un número
```

6. CUADROS DE DIÁLOGO DEL NAVEGADOR

Los navegadores permiten utilizar un objeto del sistema llamado **window**. Vamos a ver algunos métodos de este objeto.

Actualmente en una aplicación web profesional, no se usan estas técnicas al ser mucho más eficiente y versátil el uso de controles de formulario y manejo de componentes para informar al usuario y pedirle datos, pero es un método muy fácil para poder empezar a crear pequeños programas en JavaScript y probar las funcionalidades básicas del lenguaje.

6.1. MENSAJES DE ALERTA

Los mensajes de alerta son cuadros que muestran información al usuario. Lo que muestran es un texto que aparece en un pequeño cuadro de diálogo:

```
<!DOCTYPE html>
<html lang=”es”>
<head>
    <meta charset=UTF-8>
    <title>Mensaje de alterta</title>
</head>
<body>
    <script>
        alert(“Hola”);
    </script>
</body>
</html>
```

El resultado es:

Por supuesto es posible utilizar para el mensaje todo tipo de strings:

```
<!DOCTYPE html>
<html lang=”es”>
<head>
    <meta charset=UTF-8>
    <title>Mensaje de alterta</title>
</head>
<body>
    <script>
        let nombre="Gema";
        alert(`Mi hombre es ${nombre}`);
    </script>
```

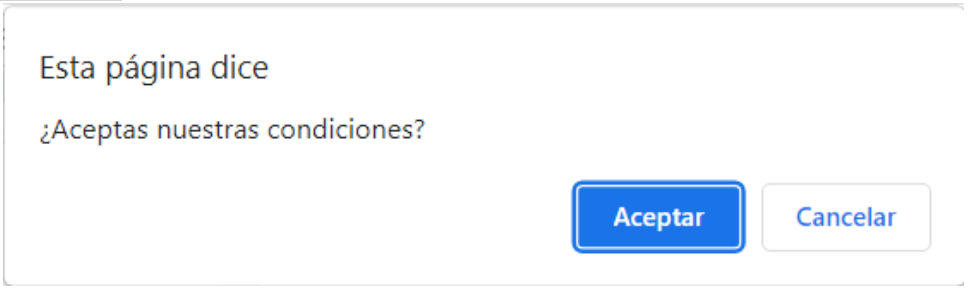
```
</body>
</html>
```

Resultado:

6.2. CUADROS DE CONFIRMACIÓN

Otro tipo de cuadros interesante es el de confirmación. El método encargado de esta tarea es **confirm**. Al igual que ocurre con **alert**, confirm requiere de un texto que será el mensaje que mostramos al usuario. El cuadro es distinto porque admite aceptar o no el mensaje. Para saber lo que ha elegido el usuario, el resultado de este método es el valor true o false. Ejemplo:

```
<script>
   let resultado=confirm("¿Aceptas nuestras condiciones?");
   console.log(resultado);
</script>
```



Después se mostrará por consola el valor true si el usuario acepta el cuadro y false si cancela el mensaje.

6.3. CUADROS DE ENTRADA DE TEXTO

El cuadro más completo es **prompt**. Es un cuadro que muestra un mensaje y, además, recoge lo que el usuario escriba en él. Ejemplo:

```
<script>
        let resultado=prompt("¿Aceptas nuestras condiciones?");
        alert(`Tu nombre es ${resultado}`);
    </script>
```

Inicialmente aparecerá este cuadro:

Después un cuadro de alerta muestra el nombre en el mensaje:

El método prompt admite indicar un valor por defecto. Por ejemplo:

```
let resultado=prompt("Escribe tu edad",18);
```

Se mostrará un cuadro en el que, ya aparece el valor 18.

Recoger valores del usuario es fundamental para poder programar. Aunque usar este tipo de cuadros, no es el método ideal para recoger valores, nos va a resultar muy útil utilizar estos métodos para entender el funcionamiento de las estructuras de control de JavaScript.

- ❑ **ACTIVIDAD 3** --- LECTURA DE NÚMEROS ---

7. CONTROL DEL FLUJO DEL PROGAMA

7.1. INTRODUCCIÓN

Hasta ahora las instrucciones que hemos visto, se ejecutan de manera secuencia. Podemos saber lo que hace el programa leyendo las líneas de izquierda a derecha y de arriba abajo.

Las instrucciones de control de flujo permiten alterar esta forma de ejecución. El uso de esas instrucciones propicia que haya líneas de código que se ejecutarán o no dependiendo de una condición. Incluso lograremos que se puedan repetir continuamente una serie de instrucciones mientras se siga cumpliendo la condición que deseemos.

Esa condición se construye utilizando lo que se conoce como expresión lógica. Una expresión lógica es cualquier tipo de expresión válida en JavaScript que proporcione un resultado booleano. Las expresiones lógicas se construyen usando variables booleanas, o bien, escribiendo expresiones que combinen los operadores relacionales y lógicos que ya hemos visto.

7.2. NÚMEROS ALEATORIOS

Los números aleatorios en JavaScript requieren del uso del objeto **Math**. Su uso permite que las aplicaciones realicen acciones imprevisibles y son fundamentales para programar juegos de todo tipo o simulaciones.

Para generar números aleatorio utilizaremos el método **random** de la siguiente forma:

```
Math.random()
```

Esta expresión da como resultado un número decimal entre 0 y 1.

```
Math.random()*2  //El resultado es un número entre 0 y 2 (siendo casi imposible que salga el 2)
Math.random()*4+6  //El resultado es un número entre 6 y 10. 4 porque el rango entre 6 y 10 es 4.
```

Si deseamos número enteros, debemos ayudarnos de la función **parseInt**, puesto que esta función elimina los decimales. Calcular un número aleatorio del 0 al 10 (incluidos ambos) sería:

```
ParseInt(Math.random()*11)
```

Si deseamos número enteros entre el 6 y el 10, la expresión sería:

```
ParseInt(Math.random()*5)+6
```

Math.random()*5 números entre el 0 y el 5 (sin llegar a 5)

- ❑ **ACTIVIDAD 4** --- FONDO ALEATORIO ---

7.3. INSTRUCCIÓN IF

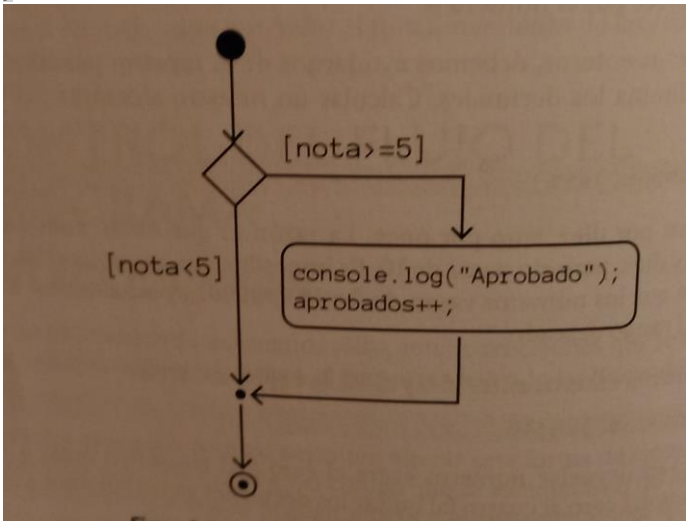
7.3.1. SENTENCIA CONDICIONAL SIMPLE

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de instrucciones en caso de que la expresión lógica sea verdadera. Si la expresión tiene un resultado falso, no se ejecutarán dichas instrucciones.

```
If (expresión lógica){
instrucciones
...
}
```

Las llaves se requieren solo si hay varias instrucciones. Ejemplo:

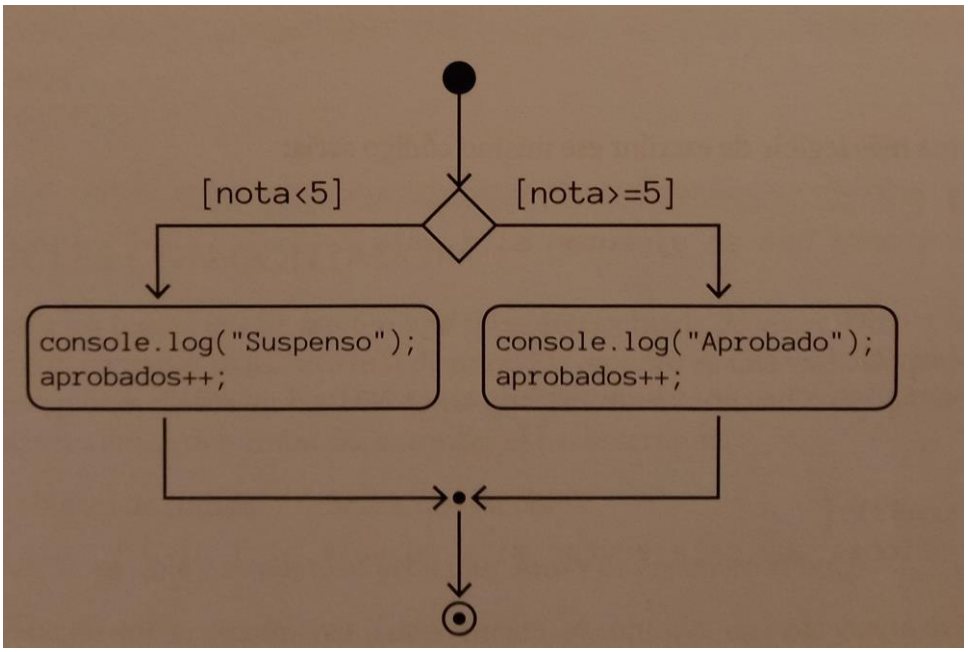
```
If(nota>=5){
console.log(“Aprobado”);
}
```



```
If (expresión lógica){
instrucciones
...
}
else {
        Instrucciones
        ...
}
```

Ejemplo de sentencia if-else:

```
If (nota>5){
console.log(“Aprobado”);
        aprobados++;
else {
        console.log(“Suspendido”);
        suspensos++;
}
```



7.3.3. ANIDACIÓN

Dentro de una sentencia **if** se puede colocar otra sentencia **if**. A esto se le llama **anidación** y permite crear programas donde se valores expresiones complejas. La nueva sentencia puede ir tanto en la parte **if** como en la parte **else**.

Las anidaciones se utilizan muchísimo al programar. Solo hay que tener en cuenta que siempre se debe cerrar primero el último if que se abrió. Es muy importante tabular el código correctamente para que las anidaciones sean legibles.

El código puede ser:

```
If (x==1) {
        instrucciones;
        ...
}
else {
        If (x==2) {
                instrucciones;
                ...
        }
        else {
If (x==3) {
instrucciones;
...
                }
        }
}
```

Otra forma más legible de escribir este mismo código sería:

```
If (x==1) {
        Instrucciones que se ejecutan si x==1;
        ...
}
else If (x==2) {
        instrucciones que se ejecutan si x!=1 y x==2;
        ...
}
else If (x==3) {
instrucciones que se ejecutan si x!=1 y x!=2 y x==3;
...
 }
else {
Instruciones que se ejecutan si x!=1 y x!=2 y x!=3
}
```

Lo cual da lugar a la llamada instrucción **if-else-if**

- ❑ **ACTIVIDAD 5** --- CÁLCULO DE SALARIO ---

7.4. INSTRUCCIÓN WHILE

7.4.1. BUCLE WHILE

La instrucción **while** permite crear bucles. Un bucle es un conjunto de sentencias que se repiten mientras se cumpla una determinada condición. Los bucles **while** agrupan instrucciones que se ejecutan continuamente hasta que una condición que se evalúa sea falsa.

La condición se mira antes de entrar dentro del while y cada vez que se termina de ejecutar las instrucciones del while.

Sintaxis:

```
while (expresión lógica) {
        sentencias que se ejecutan si la condición es verdadera
}
```

El programa se ejecuta siguiendo estos pasos:

1. Se evalúa la expresión lógica
2. Si la expresión es verdadera ejecuta las sentencias, si no el programa abandona la sentencia **while**
3. Tras ejecutar las sentencias, volvemos al paso 1

Ejemplo (escribe números del 1 al 1000):

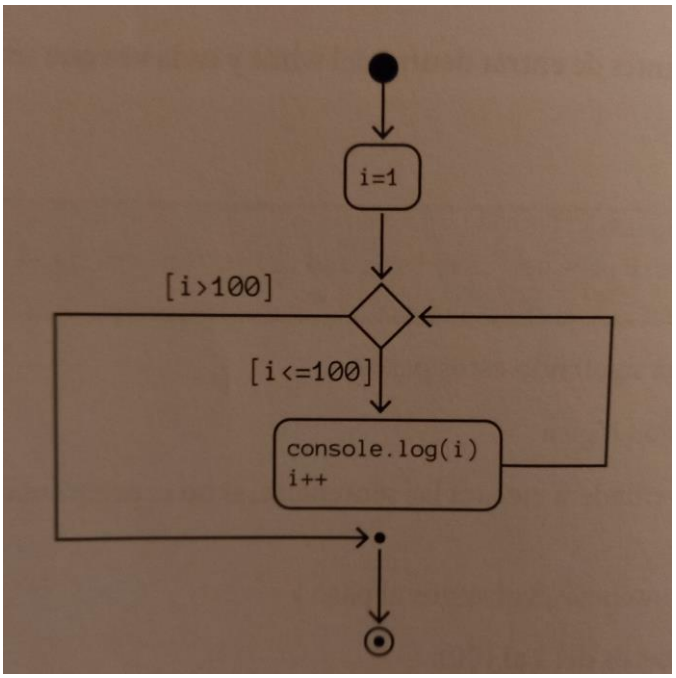
```
var i=1;
while (i<=100){
        console.log(i);
        i++
}
```

3.1.1. BUCLES CON CONTADOR

Se llaman así a los bucles que se repiten una serie determinada de veces. Dichos bucles están controlados por un contador (o incluso más de uno). El contador es una variable que va variando su valor (de uno en uno, de dos en dos, … o como queramos) en cada vuelta del bucle. Cuando el contador alcanza un límite determinado, entonces el bucle termina.

En todos los bucles de contador necesitamos saber:

1. Lo que vale la variable contador al principio. Antes de entrar en el bucle.
2. Lo que varía (lo que se incrementa o decrementa) el contador en cada vuelta del bucle.
3. Las acciones a realizar en cada vuelta del bucle.
4. El valor final del contador. En cuanto se rebase el bucle termina. Dicho valor se pone como condición del bucle, pero a la inversa; es decir, la condición mide el valor que tiene que tener el contador para que el bucle se repita y no para que termine.



Ejemplo:

```
let i=10; //valor inicial del contador, empieza valiendo 10
        /* condición del bucle, mientras i sea menor de 200, el bucle se repetirá, cuando i rebase este
        valor, el bucle termina */
while (i<= 200){
console.log(i)
        /* en cada vuelta del bucle, simplemente se escribe el valor del contador por la consola */
        i+=10;
        /* variación del contador, en este caso cuenta de 10 en 10 */
}
/* al final, el bucle escribe por consola
10
20
30
...
y así hasta 200 */
```

3.1.2. BUCLES DE CENTINELA

Es el segundo tipo de bucle básico. Una condición llamada **centinela**, que puede ser desde una simple variable booleana hasta una expresión lógica más compleja, sirve para decidir si el bucle se repite o no. De modo que cuando la condición lógica se incumpla, el bucle termina.

Ejemplo:

```
let salir=false; /* en este caso el centinela es una variale booleana que inicialmente vale falso */
let n;
while (salir==false){
/* condición de repetición: que salir siga siendo falso.
este es el centinela
```

```
        también se podía haber escrito simplemente: while (!salir)
        */
n=parseint(Math.random()*5)+1;  // lo que se repite en el bucle:
console.log(i);                 // calcular un número aleatorio de 1 a 5 y escribirlo por consola
salir= (i%7==0); /* el centinela vale verdadero si el número es múltiplo de 7 */
}
```

Comparando los bucles de centinela con los de contador, podemos señalar estos puntos:

- Los bucles de contador se repiten un número concreto de veces, los bucles centinela no.
- Un bucle contador podemos considerar que es seguro que finalice, el de centinela puede no finalizar si el centinela jamás varía su valor (aunque, si está bien programado, alguna vez lo alcanzará)
- Un bucle contador está relacionado con la programación de algoritmos basados en serie.

Un bucle podría ser incluso mixto: de centinela y contador. Por ejemplo, un programa que escriba números de 1 a 500 y que se repita hasta que llegue a un múltiplo de 7, pero que como mucho se repita 5 veces. Sería:

```
let salir=false; //centinela
let n;
let i=1; //contador
while (salir == false && i<= 5){
        n = parseint(Math.random() * 500 +1);
        console.log(n);
        i++;
        salir = (n% 7== 0);
}
console.log("último número "+n)
```

7.5. BUCLE DO...WHILE

La única diferencia respecto a la anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir, el bucle al menos se ejecuta una vez. Los pasos son:

1. Ejecutar sentencias
2. Evaluar expresión lógica
3. Si la expresión es verdadera volver al paso 1, si no continuar fuera del while.

```
do {
        instrucciones;
} while (expresión lógica)
```

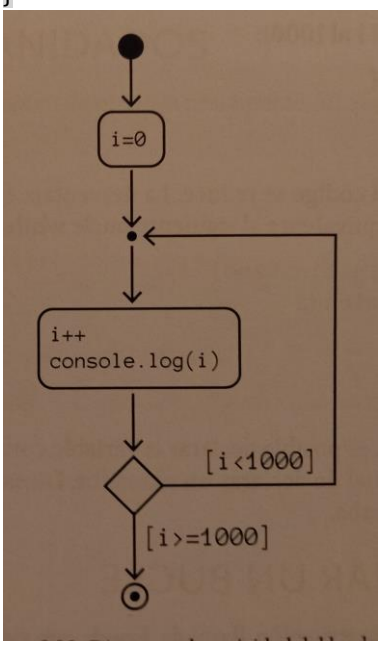
Ejemplo de contar de uno a 1000:

```
let i=0;
do {
        i++;
        console.log(i);
} while (i< 1000);
```

Se utiliza cuando sabemos al menos que las sentencias del bucle se van a repetir una vez (en un bucle **while** puede que incluso no se ejecuten las sentencias que hay dentro del bucle si la condición fuera falsa, ya desde un inicio.

De hecho, cualquier sentencia **do..while** se puede convertir en while. El ejemplo anterior se puede escribir usando la instrucción while, así:

```
let i=0;
i++;
console.log(i);
while (i< 1000) {
        i++;
        console.log(i);
}
```



- ❑ **ACTIVIDAD 6** --- JUEGO DE ADIVINAR NÚMERO ---

7.6. BUCLE FOR

Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador.

Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Sintaxis:

```
for (inicialización; condición; incremento) {
        sentencias
}
```

Las sentencias se ejecutan mientras la condición asociada sea verdadera. Además, antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir, el funcionamiento es:

1. Se ejecuta la instrucción de inicialización
2. Se comprueba la condición
3. Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bucle **for**.
4. Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2.

Ejemplo de contar números del 1 al 1000:

```
for(let i=1;i<=1000;i++){
        console.log(i);
}
```

La ventaja que tiene es que el código se reduce. La desventaja es que el código es menos comprensible. El bucle anterior es equivalente al siguiente bucle while:

```
var i=1; //sentencia de inicialización
while (i<= 1000) { //condición
        console.log(i);
        i++; // incremento
}
```

Es posible declarar la variable contadora dentro del propio bucle for. De hecho, es la forma habitual de declarar un contador. De esta manera se crea una variable que muere en cuanto el bucle termina.

7.7. ABANDONAR UN BUCLE

JavaScript proporciona una instrucción llamada break que permite abandonar de golpe un bucle:

```
let m=parseInt(Math.random()*10)+1;
for (let i=1;i<=1000;i++){
console.log(`i=${i}, m=${m}`);
if(m==10){
            break;
        }
        m=parseInt(Math.random()*10)+1;
}
```

Posible salida:

```
i=1, m=7
i=2, m=1
i=3, m=8
i=4, m=10
```

Este extraño bucle crea un contador (i) que empieza valiendo 1 y termina valiendo 1000 y cuenta de 1 en 1. Sin embargo, otra variable llamada m, calcula un número aleatorio del 1 al 10. La instrucción if comprueba si el número aleatorio calculado realmente es el 10, de ser así, interrumpe el bucle mediante la instrucción break.

Esta instrucción puede ser un atajo muy interesante para salir de un bucle de forma creativa, pero también, si se usa demasido y con poco cuidado, puede producir malos hábitos. Usar break crea código desestructurado, en el que es difícil reconocer con facilidad cuál es su flujo de trabajo.

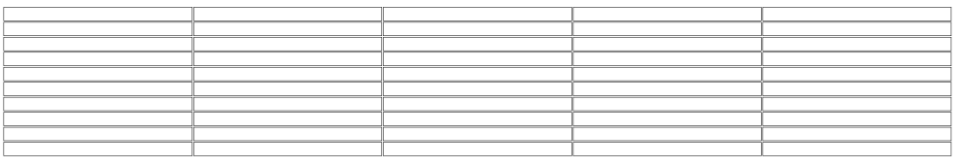
7.8. BUCLES ANIDADOS

Es posible crear un bucle dentro de otro, en una operación que se llama anidamiento. Ejemplo:

```
<!DOCTYPE html>
<html lang=”es”>
<head>
    <meta charset=UTF-8>
    <title>Document</title>
    <style>
        table{
            border-collapse: cllapse;
            width: 100%;
        }
        td{
            border: 1px solid black;
        }
    </style>
</head>
<body>
<table>
    <script>
    for(let i=1;i<=10;i++){
        document.write("<tr>");
            for(let j=1;j<=5;j++){
                document.write("<td> </td>")

            }
        document.write("</tr>");
    }
    </script>
</table>
</body>
</html>
```

El resultado de este programa es el siguiente:



El primer bucle **for** se repite 10 veces, su contador es la variable i. Cada vez que se repite se escribe la etiqueta de inicio de fila de la tabla (tr) y después aparece el segundo bucle que se repite 5 veces. Cada una de estas 5 veces, escribe una celda vacía. Finalmente se cierra la fila. Se escriben por lo tanto, 10 filas con 5 columnas en cada fila.

Los bucles anidados son muy potentes e interesantes. Por supuesto, no solo se pueden anidar bucles **for**, también se puede anidar cualquier otro tipo de bucle.

- **ACTIVIDAD 7** --- TRIÁNGULO DE ASTERISCOS ---