Introducción 1

Javascript es un lenguaje de programación utilizado para crear pequeños programas encargados de realizar acciones dentro del ámbito de una página web. Con Javascript dotaremos de interactividad a nuestras paginas. Además podemos crear efectos especiales, controlar las diferentes partes del navegador, validar entradas de datos..., en resumen, aumentar la funcionalidad de nuestras páginas web.

Javascript es el siguiente paso, después del HTML y CSS, que puede dar un programador de la web que decida mejorar sus páginas y la potencia de sus proyectos. Es un lenguaje de programación relativamente sencillo (sobre todo en comparación con los lenguajes más populares para desarrollar aplicaciones de escritorio) y pensado para hacer las cosas con rapidez, a veces incluso, con ligereza. Es por eso que, las personas que no tengan una experiencia previa en la programación podrán aprender este lenguaje con relativa facilidad y utilizarlo en toda su potencia con sólo un poco de práctica.

Es el navegador del cliente (el que visualiza la página) el encargado de interpretar las instrucciones Javascript y ejecutarlas para realizar todo lo que le indiquemos, de modo que el mayor recurso, y tal vez el único, con que cuenta este lenguaje es el propio navegador.

Javascript es un lenguaje con muchas posibilidades, permite la programación de pequeños scripts, pero también desarrollos más grandes, orientados a objetos, con funciones, estructuras de datos complejas, etc. Toda esta potencia de Javascript se pone a disposición del programador, que se convierte en el verdadero dueño y controlador de cada cosa que ocurre en la página.

En este manual vamos a tratar de acercarnos a este lenguaje en profundidad y conocer todos sus secretos y métodos de trabajo. Al final de los apuntes seremos capaces de controlar la página web y decidir la mejor estrategia para completar los problemas u objetivos que nos hayamos planeado.

1.1 Breve Historia

A principios de los años 90, la mayoría de usuarios que se conectaban a Internet lo hacían con módems a una velocidad máxima de 28.8 kbps. En esa época, empezaban a desarrollarse las primeras aplicaciones web y por tanto, las páginas web comenzaban a incluir formularios complejos.

Con unas aplicaciones web cada vez más complejas y una velocidad de navegación tan lenta, surgió la necesidad de un lenguaje de programación que se ejecutara en el navegador del usuario. De esta forma, si el usuario no rellenaba correctamente un formulario, no se le hacía esperar mucho tiempo hasta que el servidor volviera a mostrar el formulario indicando los errores existentes.

Brendan Eich, un programador que trabajaba en Netscape, pensó que podría solucionar este problema adaptando otras tecnologías existentes (como *ScriptEase*) al navegador Netscape Navigator 2.0, que iba a lanzarse en 1995. Inicialmente, Eich denominó a su lenguaje *LiveScript*.

Posteriormente, Netscape firmó una alianza con Sun Microsystems para el desarrollo del nuevo lenguaje de programación. Además, justo antes del lanzamiento Netscape decidió cambiar el nombre por el de JavaScript. La razón del cambio de nombre fue exclusivamente por marketing, ya que Java era la palabra de moda en el mundo informático y de Internet de la época.

La primera versión de JavaScript fue un completo éxito y Netscape Navigator 3.0 ya incorporaba la siguiente versión del lenguaje, la versión 1.1. Al mismo tiempo, Microsoft lanzó JScript con su navegador Internet Explorer 3. JScript era una copia de JavaScript al que le cambiaron el nombre para evitar problemas legales.

Para evitar una guerra de tecnologías, Netscape decidió que lo mejor sería estandarizar el lenguaje JavaScript. De esta forma, en 1997 se envió la especificación JavaScript 1.1 al organismo ECMA (*European Computer Manufacturers Association*).

ECMA creó el comité TC39 con el objetivo de *"estandarizar de un lenguaje de script multiplataforma e independiente de cualquier empresa"*. El primer estándar que creó el comité TC39 se denominó **ECMA-262**, en el que se definió por primera vez el lenguaje ECMAScript.

1.2 A quien va dirigido este manual

En este manual vamos a abordar el lenguaje Javascript desde lo más básico a cosas realmente complejas. Es por eso que, inicialmente, tan solo es necesario tener conocimientos de HTML y CSS para poder seguirlo.

No es necesario tener nociones de programación, aunque es muy recomendable. De esta forma, el lector que ya haya programado en otros lenguajes tipo C, JAVA, PHP... encontrará mucho más sencilla la materia que se explica en los primeros temas.

Como se puede comprobar en el indice, el libro está divido en varios bloques:

- **Parte 1- Javascript Básico:** en esta se asume que el lector no sabe nada de programación y se explica toda la parte básica sobre el desarrollo con Javascript. Abarca desde el concepto de 'variable' hasta los elementos fundamentes del DOM y del BOM.

- **Parte 2- Javascript Avanzado**: esta parte está pensada para aquellos lectores que quieran ampliar sus conocimientos sobre Javascript y es continuación directa del bloque anterior. Aprenderemos a manejar correctamente el DOM, trataremos en profundidad los eventos, cookies... y explicaremos algunas de las nuevas mejoras de Javascript asociadas a la tecnología HTML5.

Nota: Este manual engloba y amplía el contenido oficial del módulo **0612 Desarrollo Web en Entorno Cliente del Ciclo** de *Grado Superior Desarrollo de Aplicaciones Web* (DAW), tal y como se indica en el *Real Decreto 686/2010 de 20 de Mayo de 2010* y en la *Orden del 16 de Junio de 2011* de la Conserjería de Educación de Andalucía

Comenzando 2

El lenguaje Javascript tiene una sintaxis muy parecida a lenguajes derivados de C (JAVA, C++, PHP...) por estar basado en él. Es por eso que, si el lector conoce alguno de estos lenguajes se podrá manejar con facilidad con el código. De todos modos, en los siguientes capítulos vamos a describir toda la sintaxis con detenimiento, por lo que aquellos lectores que no tengan conocimiento de programación, no van a atener ningún problema.

2.1 Formas de incluir JavaScript

Existen distintos modos de incluir lenguaje JavaScript en una página. En este manual vamos a empezar enseñando la forma más adecuada de hacerlo para que nuestro código salga ordenado.

Por supuesto, no quiere decir que esa sea la forma más eficiente ni, como ya se ha comentado, la única. Pero eso es algo que se debatirá más adelante en el manual.

La forma correcta de incluir Javascript en una página HTML es utilizando la etiqueta **<script>** en el documento. Se pueden incluir tantas etiquetas **<script>** como se quiera en un documento.

El formato más común es el siguiente:

```
<script type="text/javascript">
    código JavaScript
</script>
```

En HTML5 también podemos usar

```
<script>
    código JavaScript
</script>
```

La ahora sería ¿y en que parte del documento HTML colocamos estas etiquetas? Como hemos comentado antes, ahora mismo nos vamos a preocupar de que todo nuestro código quede ordenado, por tanto vamos a colocar nuestras etiquetas `<script>` en la cabecera del documento (`<head>`)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Titulo de la página</title>
        <!-- etiquetas META -->
        <!-- enlaces a archivos CSS -->
        <!-- enlaces a recursos (pej: favicon) -->
        <script type="text/javascript">
            código JavaScript
        </script>
    </head>
    <body>
         contenido de body (código HTML)
    </body>
</html>
```

Importante: fijate que en el interior del cuerpo del documento `<body>` sólo hay etiquetas HTML con sus atributos. Aunque es posible hacerlo, nosotros NO colocamos etiquetas `<script>` en el `<body>` porque, a la larga, generan código difícil de depurar, mantener y cambiar.

Existe otra forma aún más adecuada de utilizar JavaScript en nuestra página: Haciendo uso del atributo **`src`** de la etiqueta **`<script>`** .Este atributo permite incluir un archivo externo que sólo contiene código JavaScript y que será incluido en la página.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Titulo de la página</title>
        <!-- etiquetas META -->
        <!-- enlaces a archivos CSS -->
        <!-- enlaces a recursos (pej: favicon) -->
        <script type="text/javascript" src="ruta relativa
                        archivo.js></script>
    </head>
    <body>
         contenido de body (código HTML)
    </body>
</html>
```

Es importante destacar los siguientes puntos:

- Aunque en esta forma no tiene nada en su interior, la etiqueta `<script>` debe cerrarse.
- Se enlaza un archivo externo, por tanto aparte de la página HTML debemos crear el archivo .js con el código que necesitemos.
- El archivo externo que se enlaza es un archivo del texto que contiene SOLO código JavaScript (nada de HTML o CSS) , y cuyo nombre acaba con la extensión `js`.

Para finalizar, podemos incluir código JavaScript como respuesta a algún evento. Pej:

```
<input type="submit" value="Aceptar" onclick="alert('Datos Borrados');" >
```

Aunque esto es algo que veremos más detenidamente en tema relativo a los eventos.

2.2 Ocultar código script

Aunque actualmente todos el 95% de los navegadores web entienden JavaScript , no está de más aprender a como evitar fallos de visualización de nuestra página web por incompatibilidades antiguas o porque el usuario decida desactivar la ejecución de código JavaScript en su navegador.

Esto es así porque en los casos en los que no se interpretan los scripts, los navegadores asumen que el código de éstos es texto de la propia página web y como consecuencia, presentan los scripts en la página web como si fuera texto normal.

Para evitar que el texto de los scripts se escriba en la página cuando los navegadores no pueden entenderlo se tienen que ocultar los con comentarios HTML.

```
<script type="text/javascript">
<!—
   Código Javascript
//-->
</script>
```

Vemos que el inicio del comentario HTML es idéntico a cómo lo conocemos en el HTML, pero el cierre del comentario presenta una particularidad, que empieza por doble barra inclinada. Esto es debido a que el final del comentario contiene varios caracteres que Javascript reconoce como operadores y al tratar de analizarlos lanza un mensaje de error de sintaxis. Para que Javascript no lance un mensaje de error se coloca antes del comentario HTML esa doble barra, que no es más que un comentario Javascript (tal y como veremos).

Sin embargo, las etiquetas **<script>** no las entienden los navegadores antiguos, por lo tanto no las van a interpretar, tal como hacen con cualquier etiqueta que desconocen.

Para estos casos existe la posibilidad de indicar un texto alternativo para informarles de que en ese lugar debería ejecutarse un script y que la página no está funcionando al 100% de sus capacidades. Para ello utilizamos la etiqueta **<noscript>** y entre esta etiqueta y su correspondiente de cierre podemos colocar el texto alternativo al script.

```
<body>
  <noscript>
     Este navegador no puede ejecutar los scripts que se
     hay en la página. Te recomendamos que <b> actualices
     tu navegador a una versión más moderna </b>
   </noscript>
    . . .
</body>
```

Fijate que la etiqueta <noscript> se debe incluir en el interior de <body> (normalmente al principio). Entre etiquetas <noscript> podemos meter cualquier elemento o etiqueta de HTML.

2.3 Reglas sobre la sintaxis

La **sintaxis** de un lenguaje de programación se define como el conjunto de reglas que deben seguirse al escribir el código de los programas para que estos puedan considerarse como correctos en ese lenguaje de programación.

Las reglas básicas que definen la sintaxis de Javascript son las siguientes:

2.3.1 Comentarios

Un comentario es una parte de código que no es interpretada por el navegador y cuya utilidad radica en facilitar la lectura al programador. El programador, a medida que desarrolla el script, va dejando frases o palabras sueltas, llamadas comentarios, que le ayudan a él o a cualquier otro a leer mas fácilmente el script a la hora de modificarlo o depurarlo.

Existen **dos** tipos de comentarios en el lenguaje:

- La doble barra (//): sirve para comentar una línea de código.
- Los signos /* para empezar el comentario y */ para terminarlo: nos permiten comentar trozos de varias lineas de código.

```
<script>

   //Este es un comentario de una línea

   /*
     Este comentario se puede extender
     por varias líneas.
     Las que quieras
   */

</script>
```

2.3.2 Mayúsculas y minúsculas

En javascript se han de respetar las mayúsculas y las minúsculas. Si nos equivocamos al utilizarlas, el navegador responderá con un mensaje de error de sintaxis.

Por convención los nombres de las cosas se escriben en minúsculas, salvo que se utilice un nombre con más de una palabra, pues en ese caso se escribirán con mayúsculas las iniciales de las palabras siguientes a la primera. Pej: `resolucionHorizontal`.

También se puede utilizar letras mayúsculas en las iniciales de las primeras palabras en algunos casos, como los nombres de las clases, aunque ya veremos más adelante cuáles son estos casos.

Importante: Ya que Javascript es sensible a mayúsculas y minúsculas, no es lo mismo "Salto" que "salto" o que “sAlTo”.

2.3.3 Separación de instrucciones

Las distintas instrucciones que contienen nuestros scripts se han de separar convenientemente para que el navegador no indique los correspondientes errores de sintaxis. Javascript tiene dos maneras de separar instrucciones. La primera es a través del carácter punto y coma (;) y la segunda es a través de un salto de línea.

Por esta razón Las sentencias Javascript no necesitan acabar en punto y coma a no ser que coloquemos varias instrucciones en la misma línea.

De todos modos, es buena idea acostumbrarse a utilizar el punto y coma después de cada instrucción, pues otros lenguajes como PHP, JAVA o C# obligan a utilizarlas, con lo que, de esta forma, nos estaremos acostumbrando a escribir una sintaxis más parecida a la habitual en entornos de programación avanzados.

2.4 Como vamos a trabajar

Si consultamos manuales y cursos sobre Javascript, veremos que hay muchas formas de colocar código Javascript en nuestras páginas web. Tantas que, si no tenemos cuidado, podemos colocar tanto código en numerosos sitios provocando tal desorden que cuando queramos darnos cuenta vamos a tener una verdadera maraña de código Javascript, HTML y CSS.

Es por eso que en este manual (y hasta que se diga lo contrario) se va a seguir la siguiente forma de trabajar:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Mi pagina</title>
    <script>
        function miFuncion(){
            codigo Javascript
        }
    </script>
</head>
```

```
<body>

    <input type="button" name="boton" value="pulsa"
              onclick="miFuncion()" >

</body>
</html>
```

No es importante ahora mismo el saber qué son y cómo funcionan las sentencias `function` y `onclick`. Eso es algo que veremos a lo largo de este manual. Lo importante es quedarse con el patrón que vamos a usar para trabajar.

Los pasos realizados son los siguientes:

- En `<head>`, como ya se indicó en otro apartado de este mismo capítulo, se colocarán las etiquetas `<script>` que necesitemos.
- Dentro de las etiquetas `<script>` se va ha hacer uso de funciones. Para ellos usaremos la palabra reservada `function`. Las funciones son bloques de código ordenado y que se verán en profundidad en el capítulo 8 de este manual.

  Ahora mismo basta con saber que debemos colocar la palabra reservada `function` seguido del nombre que nosotros queramos y de dos paréntesis.
- Entre las llaves de una función es donde vamos a colocar el código Javascript que vamos a ir viendo a lo largo de los siguientes capítulos.
- Como Javascript es un lenguaje que funciona por eventos (cuando pasa algo en nuestra página), vamos a colocar un botón en nuestra página tal y como muestra el ejemplo: con el atributo `onclick` y en su interior el nombre que el hemos puesto a la función.

Con esto vamos a conseguir que cada vez que se pulse el botón, el navegador ejecute el código Javascript que le indicamos.

Aunque realmente no se programa así cuando realizamos código Javascript en nuestras páginas, es la mejor forma cuando se empieza a aprender a programar dado todo el código generado queda ordenado y bien estructurado.

Como también se indicó en el apartado 2.1 de este manual, la forma más correcta y ordenada de trabajar es colocando todo el código Javascript en un fichero aparte. De esta forma vamos a tener dos ficheros: el que tiene la estructura en HTML/CSS y el que tiene el código Javascript.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Mi pagina</title>
    <script src= "archivo.js"></script>
</head>
<body>
    <input type="button" name="boton" value="pulsa"
              onclick="miFuncion()" >
</body>
</html>
```

Estructura HTML/CSS con enlace a archivo Javascript

```
//Este archivo solo tiene codigo Javascript
//No puedo poner nada de HTML o CSS aquí

function miFuncion(){
            codigo Javascript
}
```

archivo.js

Como se puede observar esta forma es muy similar a la anterior (aunque mucho mas correcta). Tan sólo hemos 'movido' las funciones desde el interior de las etiquetas `<script>` a un fichero aparte y hemos enlazado este en la estructura HTML/CSS.

Como ya se ha indicado, estas formas de colocar código Javascript en nuestros proyectos tienen sus ventajas e inconvenientes. Sin embargo, son las más correctas de usar didácticamente hablando y son las que utilizaremos en este manual hasta que tengamos los suficientes conocimientos como para usar otras alternativas.

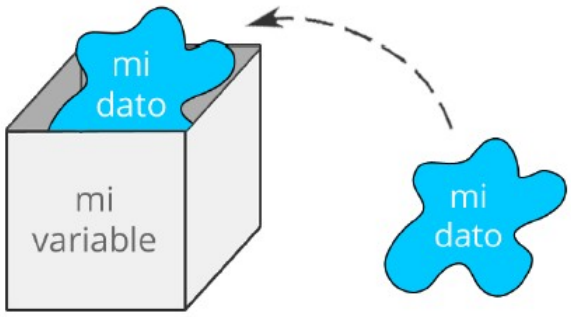
2.5 Glosario

A continuación vamos a definir una serie de conceptos básicos:

- **Script:** Cada uno de los programas o trozos de código creados con el lenguaje de programación JavaScript.
- **Sentencia:** Cada una de las instrucciones que forman un script.
- **Palabras Reservadas:** Son las palabras (en inglés) que se utilizan para construir las sentencias de JavaScript y que por tanto no pueden ser utilizadas libremente.
- **Funciones:** agrupaciones de sentencias que realizan una tarea común.

Variables 3

Lo primero que se aprende cuando se empieza a programar es el concepto de variable. Una variable no es mas que **una caja donde voy a guardar una cosa**. Tan sencillo como eso.



Cada variable va a tener un nombre único para poder hacer referencia a ellas y ver lo que guardan. El nombre se lo ponemos nosotros y podemos nombrarlas como queramos siempre que empiece por letra o que empiece por el carácter guion bajo (_) o que empiece por el carácter símbolo del dólar ($) y que no usemos palabras reservadas del lenguaje. El nombre tampoco puede contener espacios.

Nombres admitidos para las variables podrían ser

```
Edad
$paisDeNacimiento
_nombre
edad
_NOMBRE
```

Hay que tener en cuenta también que JavaScript distingue entre mayúsculas y minúsculas. Por lo que, para Javascript, `edad` y `Edad` serían dos variables diferentes.

Veamos ahora algunos nombres de variables que NO está permitido utilizar

```
12meses
tu nombre
return
pe%pe
@todes
```

3.1 Definiendo variables

"definir" en el mundo de la programación es sinónimo de "crear". Por tanto cuando definimos variables lo que estamos es "creando" esas cajitas y dejándolas listas para ser usadas.

Para definir variables en Javascript se utiliza alguna de estas dos palabras reservadas: `var` o `let` y a continuación el nombre de la variable.

```
var mivariable;
let nombre;
var numero;
let elemento2;
var Posicion;
```

La diferencia entre `var` y `let` se explica más adelante en este manual, aunque para aquellos lectores con nociones de programación podemos decir que:

- Con la palabra clave `var` se declaran variables locales y/o globales (depende del contexto) con ámbito de función.
- Con la palabra clave `let` se declaran variables local con ámbito de bloque.

Esto es algo que en este punto no es relevante (sobre todo para aquellos lectores que no saben programación), por lo que, de momento, podemos usar `var` y `let` indistintamente para declarar variables.

También se permite declarar varias variables en la misma línea, siempre que se separen por comas.

```
var operando1,operando2,edad,nombre;
```

3.2 Qué podemos guardar en variables

En una variable podemos introducir varios tipos de información, por ejemplo texto, números enteros o reales, etc.

A estas distintas clases de información se les conoce como tipos de datos. Cada uno tiene características y usos distintos, veamos cuáles son los tipos de datos de Javascript:

- **Números:** hace referencia a cualquier tipo de dato numérico: enteros o reales, positivos o negativos.
- **Cadenas:** las cadenas de caracteres representan un texto. Siempre que escribamos una cadena de caracteres debemos utilizar las comillas dobles ("). Ej: *"esto es una cadena de caracteres"*
- **Booleanos:** el tipo booleano posee únicamente dos valores: verdadero (`true`) o falso (`false`). Estos valores son muy usados en estructuras condicionales y bucles (algo que veremos más adelante).

Nota: Estos tipos de datos se explicarán con más profundidad en el capítulo siguiente.

Es importante tener presente que las variables pueden contener cosas más complicadas, como podría ser un objeto, una función, o valores predefinidos por el lenguaje (`null` o `undefined`) pero ya lo veremos más adelante.

Para los que conozcan otros lenguajes de programación fuertemente tipados (como JAVA) habrán notado que nuestras variables no están forzadas a guardar un tipo de datos en concreto y por lo tanto no especificamos ningún tipo de datos para una variable cuando la estamos declarando. Es decir, en JavaScript no es necesario indicar el tipo de la variable al definirla. Por tanto, las variables en Javascript van a poder guardar cualquier tipo de datos en cualquier momento de nuestros programas.

3.3 Cómo introducimos datos en una variable

Una vez que sabemos qué son las variables, como crearlas y qué podemos meter en ellas, lo siguiente es aprender cómo meter esos datos.

Para ello se hace uso del llamado operador de asignación, que no es nada más y nada menos que el signo igual (=).

Pare ello debemos seguir el siguiente esquema:

```
variable = valor
```

```
var nombre_ciudad, revisado;
var edad, revisado, contagios;


nombre_ciudad = "Valencia es bonita";
revisado = true;
edad = 32;
revisado = "no";
contagios = -7.33;
```

Esto quiere decir que:

- en la variable `nombre_ciudad` estamos guardando el texto *Valencia es bonita* (ojo, las dobles comillas no se guardan).
- en la variable `revisado` estamos guardando el valor verdadero (es un booleano)
- en la variable `edad` estamos guardando el número 32.
- en la variable `revisado` estamos guardando la palabra *no.*
- en la variable `contagios` estamos guardando el valor -7.33 (nótese que para definir números decimales en programación se usa el punto, no la coma para separar la parte entera de al decimal).

Importante: para "meter" un valor en una variable, dicha variable debe estar previamente definida. De lo contrario obtendríamos un error y nuestro programa no funcionaría.

También podemos asignar valores a una variable en el momento de definirla:

```
var nombre_ciudad = "Valencia es bonita";
var revisado = true;
var edad = 32;
var revisado = "no";
var contagios = -7.33;
```

En una variable solo cabe un valor. Ahora mismo, no es posible tener dos valores almacenados en una variable. Por lo que si asignamos un valor a una variable que ya tiene otro, el nuevo valor machaca al antiguo y la variable pasa a tener el valor nuevo:

```
var revisado = true;
var edad = 32;


revisado = "hola";
edad = 57;
```

En el ejemplo anterior ocurre lo siguiente:

1. Se define la variable `revisado` y se le asigna el valor true.
2. Se define la variable `edad` y se le asigna el valor 32.
3. Se le asigna a la variable revisado un nuevo valor: la palabra *hola*. Este valor, machaca al que tenía hasta ese momento (`true`)
4. Se le asigna a la variable edad un nuevo valor: 57 Este valor, machaca al que tenía hasta ese momento (32)

Como se ve en el último ejemplo, a diferencia de otros lenguajes como JAVA, podemos asignar valores de diferente tipo a nuestras variables cuando nos haga falta.

Si bien esto es "legal" en Javascript, no es recomendable a nivel de coherencia en el código y se considera una mala costumbre de programación.

Tipos de Datos 4

Como ya hemos dicho, los tipos de datos son la "clase de cosas que podemos guardar en una variable". Los tipos de datos existen en todos los lenguajes de programación, por lo que es interesen ate pararse a explicar los tipos básicos.

Para los lectores que provienen de otros lenguajes como JAVA, es importante indicar que JavasScript es un lenguaje de programación débilmente tipado, es decir, que las variables en Javascript se definen sin especificar a que tipo pertenecen (sin indicar qué voy a guardar en ellas). Además, tiene un tipado dinámico: las variables en Javascript son del mismo tipo del elemento que guarden en ese momento.

Los diferentes tipos de datos básicos que hay en Javascript son:

4.1 Tipo de datos numérico

Al contrario que ocurre en la mayoría de los lenguajes más conocidos, en este lenguaje sólo existe un tipo de datos numérico que alberga todos los diferentes números. Es decir, en el tipo numérico entran los números:

- **Enteros:** negativos y positivos. Pej: `2,3,67,-890`

- **Decimales:** como en la mayoría de lenguajes de programación, se usa el punto para separar la parte entera de la parte decimal, no la coma. Pej: `23.45, -89.000007`

Con Javascript también podemos escribir números en otras bases. Las bases son sistemas de numeración que utilizan más o menos dígitos para escribir los números.

Existen tres bases con las que podemos trabajar:

- **Base 10:** es el sistema que utilizamos habitualmente, el sistema decimal. Cualquier número, por defecto, se entiende que está escrito en base 10.
- **Base 8:** también llamado sistema octal, que utiliza dígitos del 0 al 7. Para escribir un número en octal basta con escribir ese número precedido de un 0, Pej: `045`.
- **Base 16:** o sistema hexadecimal, es el sistema de numeración que utiliza 16 dígitos, los comprendidos entre el 0 y el 9 y las letras de la A a la F, para los dígitos que faltan. Para escribir un número en hexadecimal debemos escribirlo precedido de un cero y una equis, Pej `0x3EF`.

4.2 Números especiales

En Javascript existen tres valores especiales que son considerados números:

4.2.1 Infinity y -Infinity

Representan el infinito positivo y negativo. Suelen obtenerse cuando hacemos operaciones que se salen del rango que tiene Javascript para almacenar números (64 bits), Pej: *Si intentamos hacer 10 elevado a 1000 o Dividir un numero distinto de 0 entre 0*

Al ser considerado un número, se pueden usar operadores de números para trabajar con él (esto lo veremos en el siguiente capítulo)

4.2.2 NaN

`NaN` significa "Not A Number" (No es un número). Obtendremos este valor al intentar dividir 0 entre 0, restar Infinito a Infinito, dividir un numero entre una cadena o hacer una operación numérica que produzca un resultado anómalo.

Además, este "número" tiene un comportamiento especial: es el único número que no es igual a sí mismo. [1]

1 Explicaremos esto cuando vemos el operador de igualdad.

4.3 Tipo booleano

El tipo booleano (boolean en inglés) sirve para guardar un **si** o un **no**, o dicho de otro modo, un **verdadero** o un **falso**. Se utiliza para realizar operaciones lógicas, generalmente para realizar acciones si el contenido de una variable es verdadero o falso.

Los dos valores que pueden tener las variables booleanas son **true** o **false**.

```
miBoleana = true
miBoleana = false
```

Destacar que, tanto `true`, como `false` son palabras reservadas de Javascript.

4.4 Cadena de caracteres

Este dato representa un texto independientemente de su longitud: puede ser una letra, palabras, frases o un libro entero[2].

Una cadena de caracteres puede estar compuesta de números, letras y cualquier otro tipo de caracteres y signos y siempre debe escribirse entre comillas dobles.

```
let miTexto = "Pepe se va a pescar";
miTexto = "23%%$ Letras & *--*";
```

En Javascript, como en otros lenguajes de programación, todo lo que se coloca entre comillas dobles es tratado como una cadena de caracteres independientemente de lo que coloquemos en el interior de las comillas y Javascript va a hacer caso de su contenido. Por ejemplo, no es lo mismo:

```
var num = 3;  //Esto es un numero
var num = "3"; //Esto es una cadena
```

2 A diferencia de otros lenguajes, Javascript sólo tiene este tipo de datos para guardar texto. No existe, pej, el tipo `char` como ocurre en JAVA o C++.

```
var cond = true;  //Esto es un booleano
var cond = "true"; //Esto es una cadena
```

Por tanto, si metemos algo entre comillas, ese 'algo' pasa a ser del tipo cadena y deja de ser de otro tipo.

Si necesitamos poner comillas en un texto que debe ir en una variable, lo que hay que hacer es definir el texto con comillas dobles y usar comillas simples dentro de las dobles:

```
var tex ="Como dijo el gran pensador:'Al Cesar lo que es del Cesar' ";
```

4.4.1 Caracteres de escape en cadenas de texto.

Existen una serie de caracteres especiales que sirven para cambiar comportamientos en una cadena de caracteres o texto, como puede ser un salto de línea o una tabulación.

Estos son los caracteres de escape y se escriben con una notación especial que comienza por una contra barra (una barra inclinada \ ) y luego se coloca el código del carácter a mostrar. Los caracteres de escapa más comunes son:

Salto de línea	`\n`
Comilla simple	`\'`
Comilla doble	`\"`
Tabulador	`\t`
Retorno de carro	`\r`
Avance de página	`\f`
Retroceder espacio	`\b`
Contrabarra	`\\`

Un carácter muy común es el salto de línea, que se consigue escribiendo `\n`.

```
var texto;
texto = "Con los dedos\n de las manos\n y los dedos\n de los pies..."
```

Si mostráramos por pantalla el contenido de la variable `texto` obtendríamos:

```
Con los dedos
de las manos
y los dedos
de los pies...
```

Otro uso muy habitual es colocar cuando necesitamos usar comillas doble dentro de una cadena de caracteres. Si colocamos unas comillas sin su carácter especial nos cerrarían las comillas que colocamos para iniciar la cadena de caracteres y eso provocaría error:

```
var texto = "Intento definir la división de los
"gametofitos couniformes" y sus subclases" ;
//Esto produce un error de sintáxis.
```

Las comillas adicionales las tenemos que introducir entonces con el carácter `\"`:

```
var texto = "Intento definir la división de los
\"gametofitos couniformes\" y sus subclases" ;
//Esto está correcto
```

Destacar que algunos de los caracteres mostrados probablemente no los llegarás a utilizar nunca, pues su uso está orientado a proyectos algo más complejos basados en la lectura e interpretación de cadenas de texto provenientes de diversas fuentes (introducidas por el usuario, leídas de ficheros o desde una base de datos...)

4.5 Template Strings

También llamadas *plantillas literales* (Template Literals). No es más que una nueva forma de definir cadenas de caracteres que nos aporta una serie de ventajas con respecto a la forma anterior (la forma clásica).

Para ello en vez de las comillas dobles, lo que se usa es el carácter de tilde invertida (o grave accent): ``[3]

```
var otroTexto = `Pepe se va a pescar`;
var masTexto = `23%%$ Letras & *--*`;
```

Como puede observarse en el ejemplo, tan solo se han cambiado las comillas dobles por los caracteres de tilde invertida.

Las ventajas que obtenemos al definir una cadena de caracteres de esta forma son las siguientes:

1. **Ahorrarnos caracteres de escape:** podemos usar comillas dobles o simples dentro de la cadena sin necesidad de utilizar caracteres de escape.

```
var frase =` A este chico suelen llamarle "Xino" cuando
quieren buscarle 'las cosquillas'`;
```

2. **Definir la cadena en partes:** podemos usar el salto de linea como elemento de la cadena y será respetado.

```
var queHacer =
  `comprar aceite
   comprar filetes
```

3 La tecla a la derecha de la P. Al ser un carácter tilde, de debe pulsar la tecla y luego la barra espaciadora para que aparezca el carácter solo.

```
   calentar la plancha
   freír cebolla
  `;
```

3. **Interpolación de variables:** podemos colocar el nombre de una variable dentro de la cadena usando la nomenclatura `${nombre_variable}` y automáticamente será sustituido por su valor correspondiente.

```
var nombre = "Sr. Hormiga";
var asignatura = "Programación JS";
var texto = `El profesor de la asignatura ${asignatura}
será: ${nombre}`;


console.log(texto);
```

Se obtiene la frase: *El profesor de la asignatura Programación JS será: Sr. Hormiga.*

4.6 Valores vacíos

En Javascript existen dos valores especiales que se usan para indicar la ausencia de un valor significativo: `undefined` y `null`

`undefined` es el "valor" que tendrán por defecto todas aquellas variables que se definan y a las que no se les asigne ningún valor significativo.

`undefined` y `null` significan lo mismo. El que existan dos valores para representar un mismo "valor" es simplemente un error en el diseño inicial de Javascript que no es relevante para el objetivo de este manual

4.7 Coerción de tipos

La coerción de tipos es un comportamiento muy importante que posee Javascript y que hace referencia al proceso mediante el cual JavaScript convierte el valor de una variable de un tipo a otro, normalmente de forma automática (sin que el programador lo indique).

Esto es debido al tipado débil y dinámico que posee Javascript, el cual nos permite manejar variables con menos ataduras. El problema es que, esta cualidad es un arma de doble filo si no comprendemos que es lo que está pasando al momento de operar con datos de distintos tipos.

Para entender este comportamiento son necesarios entender una serie de conceptos que aún no se han visto, por lo que explicaremos algo tan importante como es la Coerción de tipos en la parta Avanzada de este manual.

Operadores 5

Al desarrollar programas en cualquier lenguaje es muy común hacer uso de los operadores. Éstos se utilizan para realizar operaciones entre los diferentes datos almacenados en nuestras variables.

Existen operaciones más sencillas o complejas, que se pueden realizar con variables de distintos tipos de datos, como números o textos, y todas ellas hacen uso de diferentes operadores.

Un programa que no realiza operaciones con sus datos se va a limitar a hacer siempre lo mismo. Es el resultado de estas operaciones lo que hace que un programa varíe su comportamiento según los datos que obtenga.

5.1 Ejemplos de uso de operadores

Antes de entrar a enumerar los distintos tipos de operadores vamos a ver un par de ejemplos de éstos para que nos ayuden a hacernos una idea más exacta de lo que son. En el primer ejemplo vamos a realizar una suma utilizando el operador suma.

```
3 + 5
```

Esta es una expresión muy básica que hace la suma entre los dos operandos: número 3 y número 5 y devuelve un resultado. El problema es que, tal y como está planteada, no se almacena el resultado y, por tanto, se pierde.

Lo normal es capturar el resultado de una operación (sencilla o compleja) y meterlo en otra variable. Recuerda que para “meter” algo en una variable se usa el operador de asignación (=).

Por tanto la expresión correcta sería una combinación entre dos operadores, uno realiza una operación matemática (+) y el otro sirve para guardar el resultado en la variable (=).

```
miVariable = 3 + 5;
```

Los operadores se pueden clasificar según el tipo de acciones que realizan. A continuación vamos a ver cada uno de estos grupos de operadores y describiremos la función de cada uno.

5.2 Asignación

El operador de asignación es el más utilizado y el más sencillo. Ya hemos hablado de él en el capítulo anterior: se utiliza para guardar un valor específico en una variable. El símbolo utilizado es = (no confundir con el operador == que se verá más adelante):

```
var numero1 = 3;
```

A la izquierda del operador, siempre debe indicarse el nombre de una variable. A la derecha del operador, se pueden indicar otras variables, valores fijos y/u operaciones entre ellas.

```
var numero1 = 3;
var numero2 = 4;


5 = numero1;
/* Error, la asignación siempre se realiza a una variable, por
lo que en la izquierda no se puede indicar un número */


numero1 = 5;
// Ahora, la variable numero1 vale 5


numero1 = numero2;
// Ahora, la variable numero1 vale lo que vale la variable
numero2: 4
```

5.3 Operadores matemáticos

Son los utilizados para la realización de operaciones matemáticas simples como la suma, resta o multiplicación. En javascript son los siguientes:

+	Suma de dos valores
-	Resta de dos valores, también puede utilizarse para cambiar el signo de un número si lo utilizamos con un solo operando -23
*****	Multiplicación de dos valores
/	División de dos valores
%	El resto de la división de dos números (3%2 devolvería 1, el resto de dividir 3 entre 2)
++	Incremento en una unidad, se utiliza con un solo operando
--	Decremento en una unidad, utilizado con un solo operando

```
var numero1 = 10;
var numero2 = 5;

resultado = numero1 / numero2;  // resultado = 2
resultado = 3 + numero1;        // resultado = 13
resultado = numero2 - 4;        // resultado = 1
resultado = numero1 * numero 2; // resultado = 50
resultado = numero1 % numero2;  // resultado = 0
numero1++; //numero1 = 11
numero2--; //numero2 = 4;
```

5.4 Operadores de asignación especiales

Estos operadores nos sirven para acortar operaciones matemáticas sencillas y volcar el resultado en una variable. ¿Qué quiere decir esto? Vemos un ejemplo:

El operador de *asignación con suma* (+=) lo que hace es la suma de la parte de la derecha con la parte de la izquierda y guarda el resultado en la variable de la parte izquierda.

```
var num = 7;


num += 3;
//sumo lo que hay en la parte izquierda (7) con lo que hay en
la parte derecha (3) y el resultado lo guardo en la variable
num


//Por tanto, ahora num guarda el valor 10 (7 + 3 )
```

Seguro que más de un lector se ha dado cuenta de que el anterior ejemplo s epuede hacer también de la siguiente forma:

```
var num = 7;


num = num + 3;
//sumo lo que hay en la variable num (7) con el valor 3 y el
resultado lo guardo en la variable num


//Por tanto, ahora num guarda el valor 10 (7 + 3 )
```

Efectivamente, ambas formas son equivalentes.

Veamos una tabla con los operadores de asignación especiales que existen y su operación equivalente:

Operador	Equivalencia
variable **+=** valor	variable = variable + valor
variable **-=** valor	variable = variable - valor
variable ***=** valor	variable = variable * valor
variable **/=** valor	variable = variable / valor
variable **%=** valor	variable = variable % valor

Es normal que al principio no se le vea mucha utilidad a estos operadores, sin embargo, es una forma elegante de programar y son usados continuamente. Por ese motivo es bueno empezar a acostumbrarse a ellos desde este momento.

```
var num1 = 10;


num1 += 5; // num1= 15; Es como si hiciéramos num1 = num1 + 5
num1 -= 6; // num1= 4; Es como si hiciéramos num1 = num1 - 5
num1 *= 5; // num1= 50; Es como si hiciéramos num1 = num1 * 5
num1 /= 5; // num1= 2; Es como si hiciéramos num1 = num1 / 5
num1 %= 5; // num1= 0; Es como si hiciéramos num1 = num1 % 5
```

5.5 Operadores de cadenas

En Javascript tan sólo tenemos un operador para trabajar con las variables de tipo cadena de caracteres.

Esto no quiere decir que no podamos realizar otros tipos de operaciones con las cadenas de caracteres. Lo que ocurre es que, para ello, hay que usar una serie de funciones predefinidas en el lenguaje y esto es algo que se explica en la segunda parte de este manual.

Por tanto, ahora mismo nos es suficiente con conocer el *operador de concatenación de cadenas* (signo de la suma +) : Concatena dos cadenas, es decir, pega la segunda cadena a continuación de la primera.

```
cadena1 = "hola";
cadena2 = "mundo";
resultado = cadena1 + cadena2; //resultado vale "holamundo"
```

Un detalle importante que se puede ver en este caso es que el operador + sirve para dos usos distintos. Si sus operandos son números, los suma, pero si se trata de cadenas, las concatena.

Importante: Un caso que resultaría confuso es el uso del operador + cuando se realiza la operación con operadores texto y numéricos entremezclados. En este caso javascript asume que se desea realizar una concatenación y trata a los dos operandos como si de cadenas de caracteres se trataran, incluso si la cadena de texto que tenemos fuese un número. Esto se ve más fácilmente con el siguiente ejemplo.

```
let miNumero = 23;
miCadena1 = "pepe";
miCadena2 = "456";
resultado1 = miNumero + miCadena1;//resultado1 vale "23pepe"
resultado2 = miNumero + miCadena2;//resultado2 vale "23456"
miCadena2 += miNumero; //miCadena2 ahora vale "45623"
```

Como hemos podido ver, también en el caso del operador **+=**, si estamos tratando con cadenas de texto y números entremezclados, tratará a los dos operadores como si fuesen cadenas.

5.6 Operador typeof

No todos los operadores que poseen los lenguajes de programación tienen por que ser símbolos. Algunos se escriben como palabras.

Un ejemplo es el operador `typeof`. Este operador es un operador unario, es decir, actúa sobre un solo valor. Y lo que devuelve es el tipo de ese valor:

```
typeof 7.5 //dará como resultado la palabra "number"
typeof "J" //dará como resultado la palabra "string"
typeof Infinity //dará como resultado "number"
```

El operador `typeof` podemos usarlo en conjunto con los otros operadores vistos hasta ahora:

```
typeof (8+2) //number
typeof ((5+6+9+2)/4) //number
typeof ("Pablito clavó"+" un clavito.") //string
```

Como se ve en los ejemplos, es importante el uso de paréntesis en este tipo de operaciones para obtener los valores correctos.

5.7 Operadores lógicos y condicionales

Este tipo de operadores sirven para realizar operaciones que dan como resultado valores verdadero o falso.

Dado que estos valores se utilizan mayormente junto con las estructuras condicionales para tomar decisiones en nuestros programas, tiene poco sentido explicarlos en este punto del manual. Por tanto, vamos a dejar su explicación para más adelante: justo antes de hablar de las estructuras condicionales.

Interactividad 6

Hasta este momento hemos aprendido numerosos conceptos básicos de programación. La mejor manera de asentar todo lo visto hasta ahora es haciendo algunos ejercicios sencillos.

En este capítulo se van a enseñar unas herramientas básicas propias de Javascript que nos servirán, por el momento, para mostrar cosas por pantalla (tanto en la página web como en la consola del navegador) y para poder obtener valores que introduzca el usuario.

De esta forma, nuestros programas van a ganar en interactividad y vamos a poder practicar y asentar todo lo visto hasta este momento pudiendo crear programas un poco más complejos.

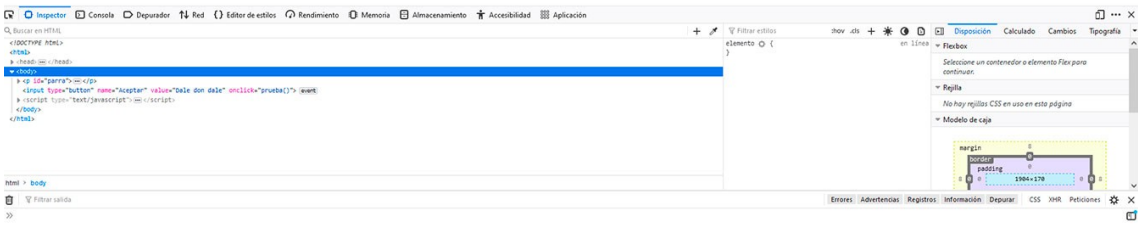
También vamos a aprender a depurar nuestros programas, es decir, a poder buscar errores en ellos. Tanto errores que impiden su funcionamiento, como errores que producen resultados incorrectos.

6.1 Herramientas para desarrolladores

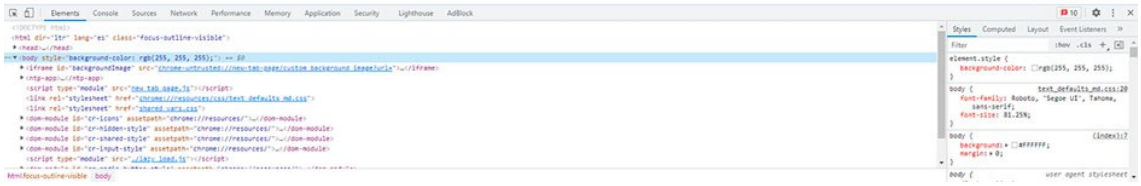
Actualmente todos los navegadores poseen un conjunto de herramientas muy completas y útiles que ayudan mucho al desarrollo de páginas y aplicaciones web. Estas son las DevTools (o herramientas para desarrolladores).

Como se ha indicado, estás herramientas están presentes en la mayoría de los navegadores que existen para PC, MAC y Linux y es posible acceder a ellas pulsando la tecla F12 cuando se tiene una página web abierta. De esta forma se mostrará una barra con contenidos que ocupará una parte de la página visualizada.

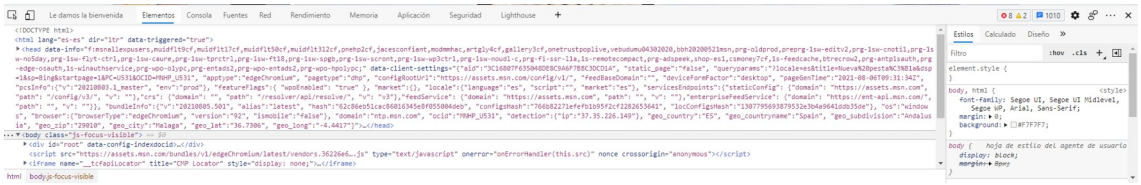
En Mozilla Firefoz la barra aparece en la parte inferior (en horizontal) de la página y tanto en Chrome como en Edge, en la parte derecha (en vertical). Esto es algo que se puede cambiar.



Herramientas de Desarrollador en Mozilla Firefox



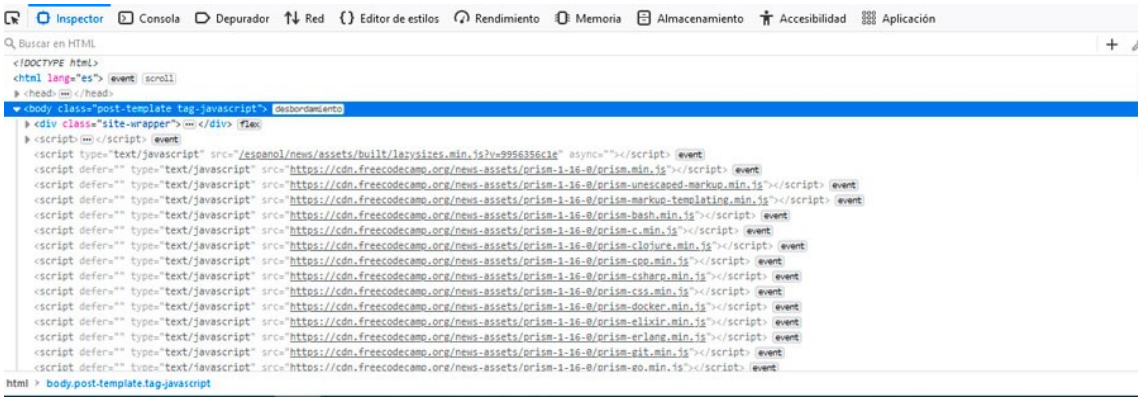
Herramientas de Desarrollador en Google Chrome



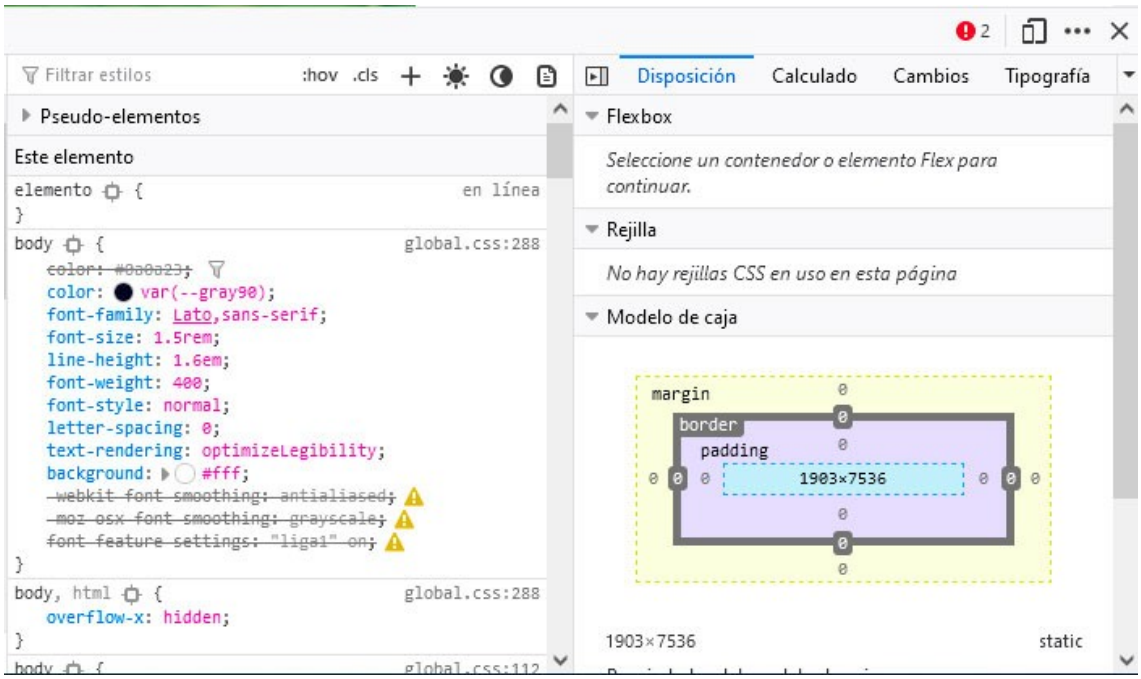
Herramientas de Desarrollador en Microsoft Edge

Aunque, como se ve en las fotos, la interfaz es parecida en todos los casos, cada navegador añade/quita/modifica alguna herramienta o funcionalidad propia. Aún así, en todos los casos nos encontramos con dos herramientas muy importantes: el inspector de elementos y la consola (las dos primeras pestañas de la barra de herramientas).

El inspector de elementos es una herramienta esencial para ver los elementos HTML de la página, comprobar todas las propiedades CSS que se le aplican a cada uno y ver de forma gráfica muy clara, lo que ocupa cada elemento en la pagina. Al ser una herramienta orientada a HTML y CSS, nos vamos a limitar a mostrar un par de imágenes de ella en este manual.



Zona donde puede inspeccionar el codigo HTML



Propiedades CSS del elemento seleccionado y lo que ocupa en la página

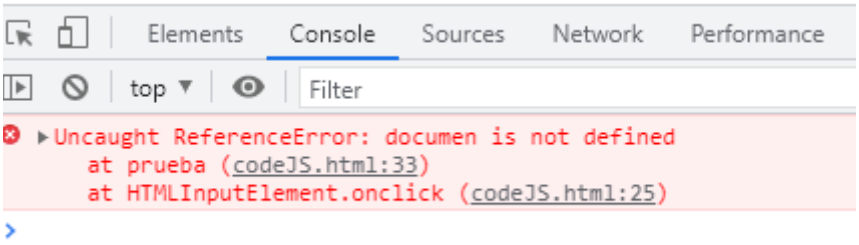
La pestañas que más se utilizan cuando se realiza desarrollo en Javascript son la **consola** y el **depurador** (o fuentes/sources en algunos navegadores).

6.2 La consola

La consola nos va a servir para recibir avisos y errores que se pueden producir al ejecutar nuestro código Javascript.

Si al intentar ejecutar nuestro código Javascript en el navegador no sucede nada, lo primero que hay que hacer es mirar si en la consola tenemos algún aviso de fallo o error.

Por ejemplo:



El mensaje que aparece en la imagen nos indica que en la línea 33 del archivo *codeJS.html* se está llamando a una función que se desconoce: `documen` (sin la t).

Es decir, está indicando que hay un error de sintaxis en el código y, por tanto, no puede seguir ejecutando nada más.

Javascript no mira primero todo el código en busca de errores y los muestra antes de ejecutarlo (hace como JAVA o C++). Por tanto, el navegador va a ir ejecutando el código hasta que encuentre un error (o acabe el programa). Al encontrar un error, lo indica y se para.

La consola también nos va a servir para mostrar resultados que se vayan produciendo en partes de nuestro código. Para ello debemos indicar en nuestros programas que queremos *escribir en la consola*.

6.2.1 Mostrar valores por consola

Para mostrar valores por consola podemos usar el objeto[4] `console`. Concretamente las siguientes herramientas:

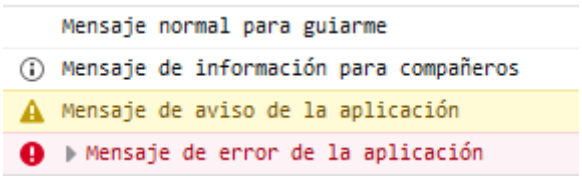
- `console.log("mensaje")`: muestra un mensaje para guiarnos mientras desarrollamos.
- `console.info("mensaje")`: muestra un mensaje para dar información a compañeros que estén desarrollando con nosotros.

4 Más adelante aprenderemos qué son los objetos. De momento podemos quedarnos en que son agrupaciones de herramientas.

- `console.warn("mensaje")`: se usa para mostrar mensajes de aviso de la propia aplicación.
- `console.error("mensaje")`: se usa para mostrar mensajes de error de la propia aplicación.

Recuerda que todas estás herramientas muestran el mensaje correspondiente por consola, pero que, como cada herramienta tiene un significado diferente, el mensaje se va a mostrar un poco diferente dependiendo de la que usemos:

```
console.log("Mensaje normal para guiarme");
console.info("Mensaje de información para compañeros");
console.warn("Mensaje de aviso de la aplicación");
console.error("Mensaje de error de la aplicación");
```



Fijate que, cuando indicamos el mensaje que se va a mostrar, este debe ser una cadena. Podemos escribirla nosotros directamente (como en el ejemplo anterior) o puede estar almacenada en una variable.

```
var m1,m2, mns;
m1 = "AVISO:";
m2 = " posible fallo en el programa.";
mns = m1+m2;

console.warn(mns);
```

6.2.2 Limpieza y orden en la consola

A poco que nuestra aplicación se vuelva medianamente compleja (usando asincronía, pej), la consola puede llenarse de avisos, errores y mensajes que pongamos.

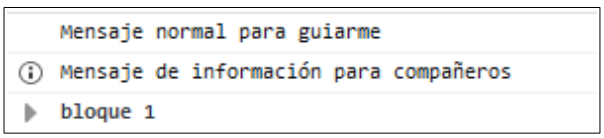
Para mantener un mínimo de orden, podemos usar las consiguientes herramientas:

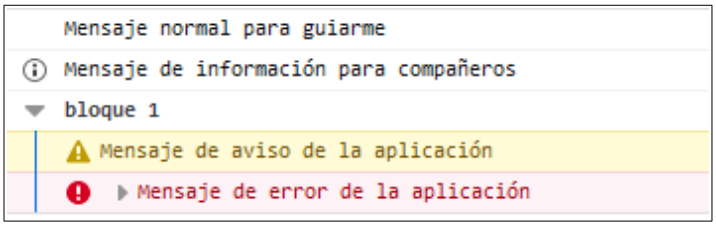
- `console.clear()`: elimina todos los mensajes que hay en la consola hasta ese momento. Tras usarse, se muestra un mensaje indicando que la consola se ha limpiado(en español o en inglés dependiendo del navegador).
- `console.groupCollapsed`: este bloque de herramientas sirve para definir grupos de mensajes en la consola. Para ello, se us ala siguiente plantilla:

```
console.groupCollapsed("nombre")
grupo de mensajes
console.groupEnd()
```

Estos grupos aparecerán colapsados (ocultos) y con el nombre indicado en `groupCollapsed`. Debe ser el usuario el que los expanda para ver los mensajes.

```
console.log("Mensaje normal para guiarme");
console.info("Mensaje de información para compañeros");
console.groupCollapsed("bloque 1");
     console.warn("Mensaje de aviso de la aplicación");
     console.error("Mensaje de error de la aplicación");
console.groupEnd();
```





6.3 Ventanas modales

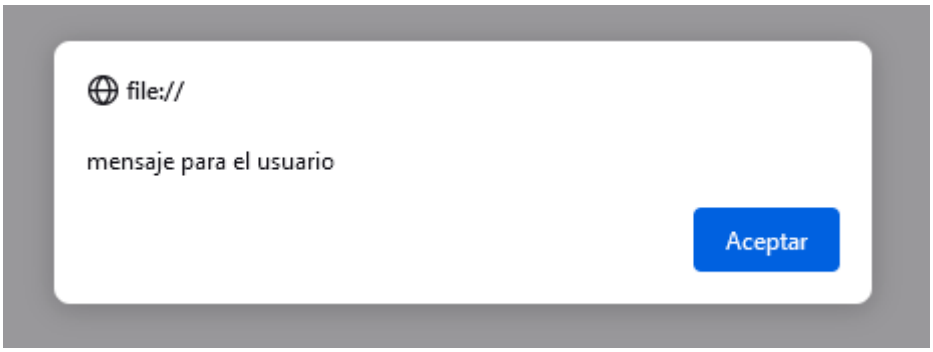
Javascript tiene predefinidas algunas herramientas que van a permitir interactuar con el usuario a través de ventanas modales[5].

Es cierto que estas herramientas están cada vez más en desuso, pero aún así, nos van a ser muy útiles en el nivel en el que nos encontramos para poder darle algo de interacción a nuestros programas.

6.3.1 Ventana de alerta

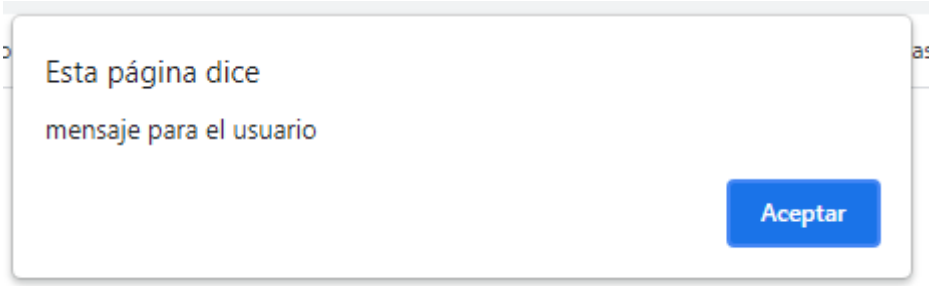
La sintaxis es: `alert("mensaje");`

Crea una ventana modal de alerta mostrando el mensaje indicado. El programa solo puede continuar cuando el usuario cierre esa ventana de aviso.



alert en Mozilla Firefox

5 Una ventana modal es una ventana que se encuentra sobre todas las demás ventanas de la misma aplicación, hasta que se cierra o se abre otra ventana modal

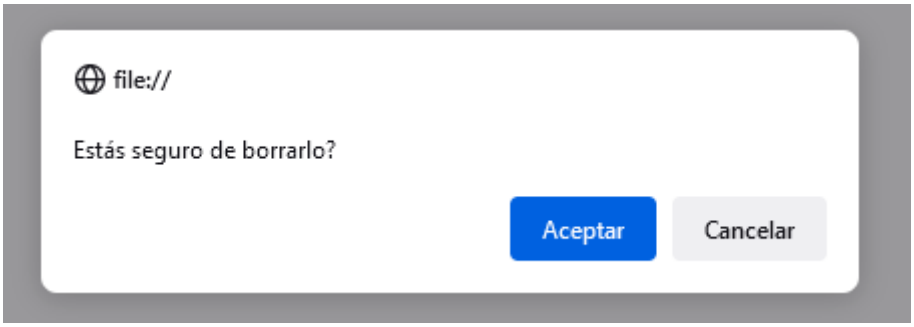


alert en Google Chrome

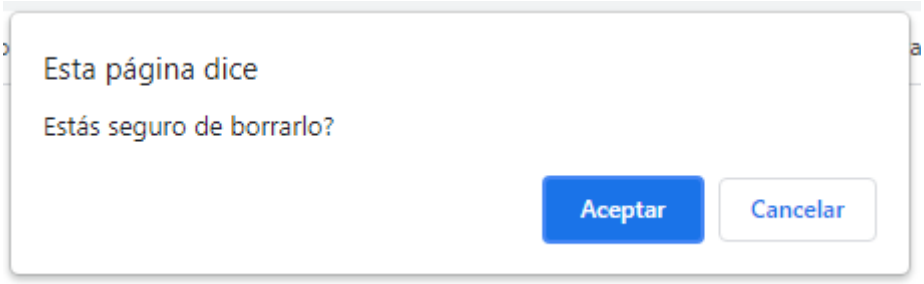
6.3.2 Ventana de confirmación

Muestra un aviso de confirmación en el que el usuario debe indicar si está de acuerdo o no pulsando el botón correspondiente (Aceptar o Cancelar).

Su sintaxis es: `confirm(pregunta para el usuario);`



confirm en Mozilla Firefox



confirm en Google Chrome

Destacar que `confirm` nos dice si acepta o no la pregunta realizada. Para ello lo que hace es devolver un valor booleano (`true` o `false`) dependiendo del botón pulsado por el usuario: Aceptar = true, Cancelar = false.

Si utilizamos `confirm` tal y como hemos mostrado, no vamos a poder saber qué ha respondido el usuario. Para solucionar esto hay que "meter" lo que devuelve `confirm` en una variable:

```
var respuesta;
respuesta = confirm("Estás seguro de borrarlo?");
console.log(respuesta);
```

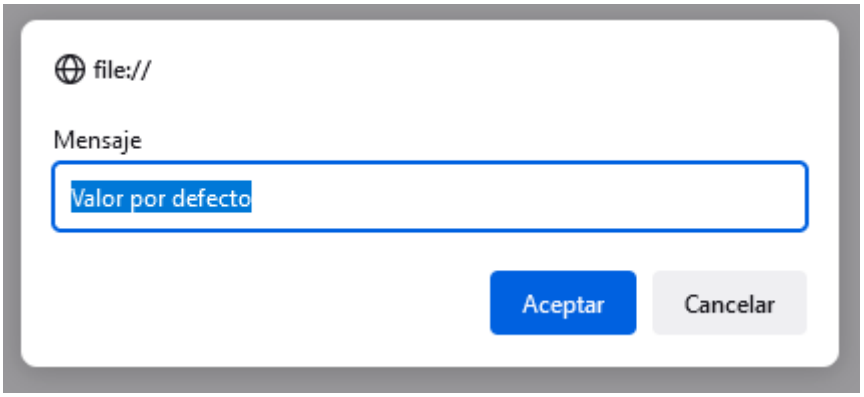
De esta forma, en la variable respuesta vamos a tener un valor booleano dependiendo de lo que elija el usuario. Más adelante veremos cómo tomar decisiones en nuestro código base a preguntas y valores booleanos.

6.3.3 Pedir valores al usuario

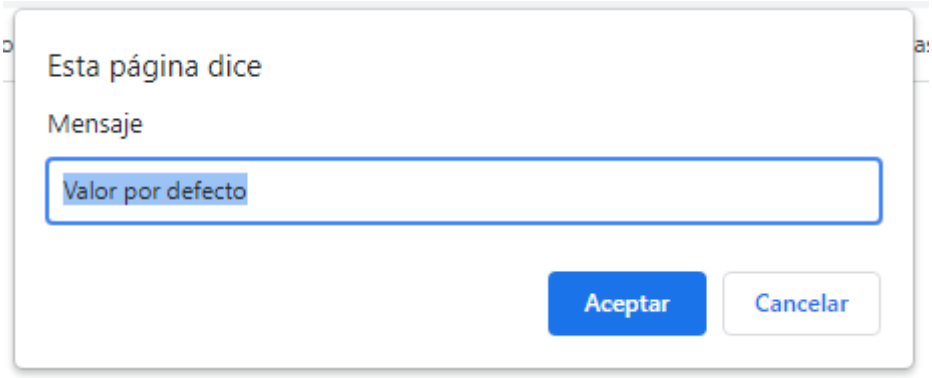
Javascript permite que se le soliciten valores al usuario a través de una ventana modal usando la herramienta `prompt`:

```
prompt(Mensaje, valor por defecto)
```

Al usar esa herramienta, se mostrará por pantalla una ventana modal con un mensaje y con un campo de texto en el que el usuario podrá escribir:



prompt en Mozilla Firefox



prompt en Google Chrome

Fijate que son necesarias dos cadenas para usar la herramienta: el mensaje que queremos mostrar y el valor por defecto que tendrá el cuadro de texto.

Si no queremos ningún valor por defecto, es obligatorio colocar una cadena vacía: `prompt(Mensaje, "")`

Al igual que ocurre con `confirm`, para poder saber qué ha introducido el usuario debemos "capturar" ese valor en una variable:

```
var usuario;
usuario = prompt("Introduce tu nombre:","");
alert("Tu nombre es:"+usuario)
```

Sin embargo, `prompt` tiene un comportamiento que hay que tener en cuenta. SI hacemos algo como esto:

```
let edad,futuro;
edad = prompt("Introduce tu edad:","");
futuro = edad + 10;
console.log("Dentro de 10 años tendrás: "+futuro);
```

y metemos el valor 33, obtenemos:

```
Dentro de 10 años tendrás: 3310
```

Esto es debido a que `prompt` siempre va a devolver una cadena de caracteres. Si sumamos una cadena de caracteres con lo que sea ¿recuerdas qué pasaba? Pues que pasaba a cadena ese "lo que sea" y lo concatenaba. Por eso obtenemos 3310.

Cuando necesitemos pasar a número lo que venga del prompt, vamos a tener que usar la función parseInt. Esta función transforma a número la cadena que se le ponga en los paréntesis:

```
let edad,arreglo, futuro;
//pedimos la edad
edad = prompt("Introduce tu edad:","");
//La pasamos a numero
arreglo = parseInt(edad);
//Operamos con ella de forma correcta
futuro = arreglo + 10;
console.log("Dentro de 10 años tendrás: "+futuro);
```

ahora sí, metiendo el valor 33, obtenemos:

```
Dentro de 10 años tendrás: 43
```

Antes de usar `edad` la pasamos a número y metemos ese resultado en otra variable (`arreglo`). Es con esa variable con la que debemos hacer las operaciones que necesitemos.

Para acabar este punto vamos a mostrar cómo hacer **más corto y limpio el código** del ejemplo anterior:

```
//pedimos la edad y la pasamos a numero antes de guardarla
var edad = parseInt(prompt("Introduce tu edad:",""));
//Operamos al mostrar
console.log("Dentro de 10 años tendrás: "+(edad+10));
```

Fijate que en una sola línea hemos definido la variable edad, hemos pedido su valor, lo hemos pasado a número y lo hemos almacenado,

Además, a la hora de mostrar el resultado, hemos realizado los cálculos en la misma función de mostrar (sea la que sea: `alert, console.log, .warn, ...`)

En esta parte son muy importantes los paréntesis. Se anima al lector a probar esta linea son los paréntesis de `(edad+10)` y a averiguar qué ocurre sin ellos. ¿Por qué ocurre eso al quitarlos ?

Estructuras condicionales 7

Hasta este punto, todos nuestros programas van ejecutando cada sentencia una debajo de otro. Este funcionamiento nos limita mucho nuestra forma de afrontar problemas en programación dado que en algún momento de nuestro programa puede interesarnos ejecutar un trozo de código u otro dependiendo de alguna condición *(Pej: si el usuario es mayor de edad, muestro un tipo de publicidad en mi página y si no lo es, muestro otra*)

Para poder realizar este comportamiento (decidir qué partes de mi programa se ejecutan en cada momento) se usan las estructuras condicionales que vamos a ver en este capítulo.

Antes de ver estas estructuras, vamos a hablar de algo que quedo pendiente en el capítulo 5: de los operadores lógicos y condicionales.

Es importante comprender bien estos operadores por que con ellos se crean, no solo las estructuras condicionales, sino los bucles que tanto se usan cuando se programa.

7.1 Operadores condicionales

Estos operadores sirven para evaluar expresiones, indicando si estas son verdaderas o falsas. Es decir, el resultado de aplicar un operador condicional es un valor booleano: `true` o `false`.

Esto puede sonar muy raro al lector que no posea conocimientos de lógica básica y algebra de Bool, pero en realidad es un concepto muy sencillo e intuitivo.
Los operadores condicionales no son más que preguntas del tipo:

¿12 es mayor que 4?
¿Lo que hay en la variable `num` *es igual a 3?*
¿La variable `edad` *tiene un valor diferente a 50?*
¿23 es mayor o igual a 100?

Es importante destacar que la respuesta a cualquiera de ese tipo de preguntas es la misma: Sí o No, verdadero o falso.

Los operadores condicionales son:

==	Comprueba si dos valores son iguales
!=	Comprueba si dos valores son distintos.
>	Mayor que, devuelve true si el primer operador es mayor que el segundo
<	Menor que, es true cuando el elemento de la izquierda es menor que el de la derecha
>=	Mayor igual.
<=	Menor igual

```
var res;
var num = 5;
var otro = 4;

res = (3 < 7); // res tendrá el valor true
res = (num <= 10); // true
res = (otro > 100); //false
res = (num == otro); //false
res = (num != otro); //true
```

***Nota**: el uso de los paréntesis es opcional, aunque el autor de este manual es un defensor de su uso ya que se obtiene claridad de lectura en expresiones complejas.*

Importante: como se comentó en el capítulo 4 de este manual, el valor `NaN` es el único número que no es igual a sí mismo:

```
NaN == NaN //false
```

También podemos comprobar que `undefined` y `null` son el mismo valor en realidad:

```
null == undefined //true
```

7.1.2 Operadores de igualdad/diferencia estricta

==	Comprueba si dos valores son estrictamente iguales.
!=	Comprueba si dos valores son estrictamente distintos.

Al igual que sucede con el operador de igualdad simple (==), el operador de igualdad estricta (===) revisa si dos operandos son iguales y devuelve `true` o `false` en cada caso.

Lo que diferencia a ambos operadores es que, el operador de igualdad estricta siempre va a devolver `false` si los operandos son de distinto tipo, cosa que no ocurre con el de igualdad simple por la coerción de datos que aplica Javascript[6].

Ocurre igual con los operandos de diferencia simple `(!=)` y diferencia estricta (`!==`): si los operandos son de diferente tipo, el operador estricto devolverá siempre `true`.

7.2 Operadores lógicos

Los operadores condicionales vistos en el apartado anterior se aplicaban sobre expresiones numéricas y obteníamos un valor booleano.

Los operadores lógicos, en vez de trabajar con números o cadenas, operan sobre valores booleanos (`true o false`) y dan como resultado otro valor booleano.

Existen numerosos operadores lógicos pero los que más vamos a usar nosotros son los siguientes:

&&	Operador AND
||	Operador OR
!	Operador NOT

6 El concepto de coerción se ve con mas detalle en la parte avanzada de este manual

7.2.1 Operador AND

En la operación lógica `AND` el resultado solamente es `true` si los dos operandos son `true`:

Primer operando	Segundo operando	Resultado
true	true	true
true	false	false
false	true	false
false	false	false

```
var res;
var num = 5;
var otro = 4;

res = (3 < 7) && (num <= 10); // res tendrá el valor true
res = (num != otro) && (otro > 100); //false
res = (num == otro) && (100 > 12); //false
res = (num < 19) && (otro > -7); //true
```

7.2.2 Operador OR

En la operación lógica `OR` el resultado es `true` si alguno de los dos operandos es `true`:

Primer operando	Segundo operando	Resultado
true	true	true
true	false	true
false	true	true
false	false	false

```
var res;
var num = 5;
```

```
var otro = 4;


res = (3 < 7) || (num <= 10); // res tendrá el valor true
res = (num != otro) || (otro > 100); //true
res = (num == otro) || (100 > 12); //false
res = (num < 19) || (otro > -7); //true
```

7.2.3 Operador NOT

Este operador solo necesita un operando, es decir, se aplica a una variable a a una expresión (no a dos como hemos visto con `AND` y `OR`). Lo que hace es cambiar el valor de esa variable/expresión por si contrario: Si vale `true`, pasa a valer `false` y viceversa.

```
var res = true;
var num = 5;
var otro = 4;


res = !res // res tendrá el valor false
res = !(num <= 10); // lo contrario de true: false
res = !(otro > 100); //lo contrario de false: true
res = !(num == otro); lo contrario de false: true
res = !(num != otro); //lo contrario de true: false
```

Es muy importante el uso de paréntesis cuando utilizamos el operador NOT para indicar a qué le estamos cambiando el valor. Ejemplo:

```
var res;
var uno = false;
var dos = true;
```

```
res = !(uno || dos);
//negamos el resultado de la operación OR y obtenemos false

res = !uno || dos;
//negamos el valor de la variable uno, hacemos el OR y
obtenemos true
```

7.3 Ejercicio simple

Vamos a plantear y a explicar un ejercicio sencillo para practicar los operadores lógicos y condicionales:

```
let res;
let num = 5;
let otro = 4;

res = !((3 < 7) && (num <= 10)); //A
res = !((num != otro) || (otro > 100)); //B
res = !((num == otro) && (100 > 12)); //C
res = !((num < 19) || (otro > -7)); //D
```

Para resolver este tipo de ejercicios siempre debemos empezar desde más adentro hacia afuera. Por eso es importante el uso de paréntesis.

Apartado A:

- ¿3 es menor que 7?: `true`
- ¿`num` (5) es menor o igual a 10?: `true`
- `true` AND `true` = `true`

- NOT `true` = `false`

Apartado B:

- ¿`num` (5) es diferente a `otro` (4)?: `true`
- ¿`otro`(4) es mayor a 100?: `false`
- `true` OR `false`= `true`
- NOT `true` = `false`

Apartado C:

- ¿`num` (5) es igual a `otro` (4)?: `false`
- ¿100 es menor que 12?: `true`
- `false` AND `true` = `false`
- NOT `false` = `true`

Apartado D:

- ¿`num` (5) es menor a 19?: `true`
- ¿`otro`(4) es mayor a -7?: `true`
- `true` OR `true` = `true`
- NOT `true` = `false`

7.4 Estructura IF

IF es la estructura de control que se suele utilizar en programación para tomar decisiones. Es un estructura que decide qué trozos de código se deben ejecutar en función de una condición.

Esta estructura puede construirse de tres formas diferentes:

7.4.1 IF simple

Esta es la forma más sencilla de usar la estructura IF. Su sintaxis es la siguiente:

```
if (condición) {
  acciones a realizar en caso de que
  se cumpla la condición
     ...
}
```

Funciona de la siguiente manera:

1. primero se evalúa la condición indicada en la cabecera de la estructura,
2. si la condición se cumple, se ejecuta la parte del código indicada,
3. en caso contrario, esa parte del código es ignorada y no se ejecuta.

```
var precio = 600;
var suma = 0;

if (precio < 1000){
     alert("He entrado en el IF");
     suma = 10;
}
alert(suma);
```

Si el valor de la variable precio es menor a 1000, muestro el mensaje *"He entrado en el IF"* y cambio el valor de la variable suma.

Destacar varias cosas:

- Para empezar vemos como con unas **llaves** engloban las acciones que queremos realizar en caso de que se cumplan o no la condición. Para nosotros, estas llaves han de colocarse siempre.
- Otro detalle que salta a la vista es el **sangrado** (margen) que hemos colocado en cada uno de los bloques de instrucciones que se ejecutan en caso de que la condición sea verdadera. Aunque este sangrado es totalmente opcional, obtenemos limpieza y claridad en el código al usarlo y, por tanto, nosotros vamos a usarlo siempre.

7.4.2 IF...ELSE

En esta forma definimos dos caminos. El primero se tomará si se cumple la condición (y se ignorará el segundo) y el segundo se tomará si la condición no se cumple (y se ignorará el primero).

La sintaxis de esta forma es la siguiente:

```
if (condición) {
  //camino A
  acciones a realizar en caso de que
  se cumpla la condición
     ...
}
else[7] {
  //camino B
  acciones a realizar en caso de que  NO
  se cumpla la condición
     ...
}
```

Funciona de la siguiente manera:

1. primero se evalúa la condición indicada en la cabecera de la estructura,
2. si la condición se cumple, se ejecuta la parte del código del camino A,
3. en caso contrario, se ejecuta la parte del código del camino B,

7 fijate que junto a ELSE no hay ninguna condición. No la pongas o te dará error.

```
if (edad >= 18) {

   //Dejo entrar en mi pagina de apuestas
   console.log("Bienvenido a mi pagina de apuestas");

   //Le regalo 100E de saldo inicial
   saldo = saldo + 100;
 }
else {
   //informo que es menor de edad
   console.log("No puedes entrar. Solo mayores de 18
años")
 }
```

En el ejemplo anterior comprobamos si la edad es mayor o igual a 18. Si es así, doy la bienvenida a mi pagina y le regalo 100€ virtuales al visitante. En caso contrario, le muestro una aviso indicando que no puede acceder.

7.4.3 IF...ELSEIF...ELSE

Con esta forma vamos a crear tantos caminos como nosotros queramos y cada uno de esos caminos va a tener su propia condición para entrar en él. Si no se cumple la condición de los caminos definidos, se ejecutará el código que hay en la parte ELSE.

La sintaxis es la siguiente:

```
if (condición A) {
  //camino A
  acciones a realizar en caso de que
  se cumpla la condición A
       ...
}
else if (condición B){
  //camino B
  acciones a realizar en caso de que
  se cumpla la condición B
       ...
}
```

```
else if (condición C){
  //camino C
  acciones a realizar en caso de que
  se cumpla la condición C
       ...
}
...
(podemos poner tantos elseif como necesitemos)
...
else {
  //camino final
  acciones a realizar en caso de que  NO
  se cumpla ninguna de las condiciones anteriores
       ...
}
```

Funciona de la siguiente manera:

1. Se evalúa la condición A
2. si la condición se cumple, se ejecuta la parte del código del camino A,
3. en caso contrario, se evaluá la condición B
4. si la condición se cumple, se ejecuta la parte del código del camino B,
5. en caso contrario, se evaluá la condición C
6. …
7. sino se cumple ninguna de las condiciones anteriores, se ejecuta la parte de código del ELSE.

A tener en cuenta:

- Las condiciones se van evaluando por orden comenzando por la primera que aparece hasta que alguna se cumpla.
- Cuando una condición se cumple, se ejecuta el código de ese camino y ya no se vuelve a evaluar ninguna otra condición de la estructura. Es decir, sólo se puede entrar por uno de los caminos de la estructura.
- Podemos definir tantos ELSEIF como nos hagan falta.
- Los ELSEIF deben llevar su condición correspondiente para poder "pasar" por ese camino.
- Como vimos en el apartado anterior, ELSE nunca lleva una condición. Su condición es que no se cumplan las otras.

A continuación vamos a mostrar el código de un programa que dependiendo del día de la semana que sea, nos saludará de una manera u otra.

Si es lunes dirá: "*Buenos dias, hoy es lunes*", si es Martes, "*Buenos días, hoy es martes*" y así sucesivamente.

```
//pedimos el dia de la semana al usuario
var dia = prompt("Que dia de la semana es?", "");

//Dependiendo del dia que sea, hago un saludo u otro
if (dia=="lunes"){
  console.log("Buenos dias, hoy es lunes");
}
else if (dia=="martes"){
  console.log("Buenos dias, hoy es martes");
}
else if (dia=="miercoles"){
  console.log("Buenos dias, hoy es miércoles");
}
else if (dia=="jueves"){
  console.log("Buenos dias, hoy es jueves");
}
else if (dia=="viernes"){
  console.log("Buenos dias, hoy es viernes");
}
else if (dia=="sabado"){
  console.log("Buenos dias, hoy es sabado");
}
else{
  console.log("Descansa, hoy es domingo");
}
```

Para ello preguntamos al usuario qué día es hoy. A continuación usando una estructura IF..ELSEIF..ELSE decidimos qué saludo se debe realizar en función del día introducido.

Dependiendo del día, se cumplirá una de las condiciones de la estructura. El flujo del programá entrará en el primer camino que se le permita (el que cumpla la condición) e ignorará le resto de posibles caminos.

Finalmente, si el día que indica el usuario no es Lunes o Martes o Miércoles o … o Sábado, se asumirá que es Domingo y se ejecutará el código de la parte ELSE.

7.5 Estructuras IF anidadas

Dentro de las llaves que engloban el código de cada uno de los caminos definidos pr una estructura IF (de cualquier tipo), podemos poner cosa de las aprendidas hasta este momento, incluidas otras estructuras IF (de cualquier tipo). Es decir, colocar estructuras IF dentro de otras estructuras IF.

De esta forma se consiguen hacer estructuras condicionales más complejas

Por ejemplo: Con un solo IF..ELSE podemos evaluar y realizar una acción u otra según dos posibilidades, pero si tenemos más posibilidades que evaluar, una forma de resolverlo sería anidar IFs para crear el flujo de código necesario para decidir correctamente.

Un ejemplo concreto: si deseo comprobar si un número es mayor menor o igual que otro, tengo que evaluar tres posibilidades distintas. Primero puedo comprobar si los dos números son iguales, si lo son, ya he resuelto el problema, pero si no son iguales todavía tendré que ver cuál de los dos es mayor.:

```
var numero1=23;
var numero2=63;

if (numero1 == numero2){
  console.log("Los dos números son iguales");
}
```

```
else{
   if (numero1 > numero2) {
      consola.log("El primer número es mayor que el segundo");
   }
   else{
      console.log("El primer número es menor que el segundo");
   }
}
```

El flujo del programa es como se ha comentado anteriormente:

- Primero se evalúa si los dos números son iguales.
- Si esto ocurre, se muestra un mensaje informando de eso.
- En caso contrario ya sabemos que son distintos, pero aun debemos averiguar cuál de los dos en mayor. Para eso se hace otra comparación para saber si el primero es mayor que el segundo.
  - Si esta comparación da resultados positivos mostramos un mensaje diciendo que el primero es mayor que el segundo.
  - En caso contrario indicaremos que el primero es menor que el segundo.

Seguro que algún se ha dado cuenta de que el ejemplo anterior podría resolverse también haciendo uso de una estructura IF..ELSEIF..ELSE e incluso sería una opción mucho mas intuitiva.

Se deja al lector la solución del ejemplo con la estructura antes mencionada y la valoración de qué ventajas e inconvenientes tiene cada forma de afrontar la solución (Anidamiento frente a estructura IF..ELSEIF..ELSE)

7.6 Estructura Switch

Esta estructura es una manera rápida y elegante de crear un bloque para tomar decisiones en función del valor que tome una variable.

En el apartado 7.4.2 pusimos como ejemplo de estructura IF..ELSEIF..ELSE un código que dependiendo del día de la semana nos saludaba de una manera u otra. Si es lunes dirá: "*Buenos dias, hoy es lunes*", si es Martes, "*Buenos días, hoy es martes*" y así sucesivamente.

```
//pedimos el dia de la semana al usuario
var dia = prompt("Que dia de la semana es?", "");

//Dependiendo del dia que sea, hago un saludo u otro
if (dia=="lunes"){
  console.log("Buenos dias, hoy es lunes");
}
else if (dia=="martes"){
  console.log("Buenos dias, hoy es martes");
}
else if (dia=="miercoles"){
  console.log("Buenos dias, hoy es miércoles");
}
else if (dia=="jueves"){
  console.log("Buenos dias, hoy es jueves");
}
else if (dia=="viernes"){
  console.log("Buenos dias, hoy es viernes");
}
else if (dia=="sabado"){
  console.log("Buenos dias, hoy es sabado");
}
else{
  console.log("Descansa, hoy es domingo");
}
```

Podemos observar que las condiciones de todos los caminos posibles son pregunta del tipo ¿es la variable igual a un valor determinado? (¿es la variable `dia` igual a “lunes”?)

Pues bien, cuando nos encontramos una estructura IF..ELSEIF..ELSE de esta forma (todas las condiciones son preguntas de igualdad), podemos usar la estructura SWITCH para resolver el problema de manera mas elegante y rápida.

Su sintaxis general de la estructura es la siguiente:

```
switch (variable) {
    case valor1:
       Sentencias a ejecutar si la expresión
       tiene como valor a valor1
       break;

    case valor2:
       Sentencias a ejecutar si la expresión
       tiene como valor a valor2
       break;

    case valor3:
       Sentencias a ejecutar si la expresión
       tiene como valor a valor3
       break;

     ...

     case valorN:
       Sentencias a ejecutar si la expresión
       tiene como valor a valorN
       break;

   default:
      Sentencias a ejecutar si el valor no
      es ninguno de los anteriores
}
```

Se mira el valor de la variable, si vale `valor1` se ejecutan las sentencias relacionadas con ese caso. Si la expresión vale `valor2` se ejecutan las instrucciones relacionadas con ese valor y así sucesivamente, por tantas opciones como deseemos. Finalmente, si no tiene ninguno de los valores indicados, se ejecuta el caso por defecto (`default`).

La palabra `break` es opcional, pero si no la ponemos a partir de que se encuentre coincidencia con un valor se ejecutarán todas las sentencias relacionadas con este y todas las siguientes. Es decir, si en nuestro esquema anterior no hubiese ningún `break` y la expresión valiese `valor1`, se ejecutarían las sentencias relacionadas con valor1 y también las relacionadas con `valor2`, `valor3` y `default`.

Para nosotros, el `break` va a ser obligatorio ponerlo de momento. Esa orden nos sirve para dejar bien claro qué sentencias se ejecutan en cada valor. Lo que si es opcional es `default`, la cual equivale (mas o menos) a un `else`.

Es fácil ver que cada bloque `case` es como si fuera un bloque `else if`. Sin embargo, no hace falta indicar ninguna condición, tan solo poner el valor que debe tener la variable si queremos que entre por ese camino.

Vamos a resolver el ejercicio anterior usando la estructura SWITCH:

```
//pedimos el dia de la semana al usuario
var dia = prompt("Que dia de la semana es?", "");

//Dependiendo del dia que sea, hago un saludo u otro
switch (dia) {
 case "lunes":
    console.log("Buenos dias, hoy es lunes");
    break;
 case "martes":
    console.log("Buenos dias, hoy es martes");
    break;
 case "miercoles":
    console.log("Buenos dias, hoy es miercoles");
    break;
 case "jueves":
    console.log("Buenos dias, hoy es jueves");
    break;
 case "viernes":
    console.log("Buenos dias, hoy es viernes");
    break;
 case "sabado":
   console.log("Buenos dias, hoy es sabado");
   break;
```

```
 default:
   console.log("Descansa, hoy es domingo");
}
```

Veamos otro ejemplo:

```
var dia = parseInt(prompt("Número de mes?", ""));

//Dependiendo del dia que sea, hago un saludo u otro
if (dia==1){
  console.log("Enero");
}
else if (dia==2){
  console.log("Febrero");
}
else if (dia==3){
  console.log("Marzo");
}
else if (dia==4){
  console.log("Abril");
}
else if (dia==5){
  console.log("Mayo");
}
else if (dia==6){
  console.log("Junio");
}
else if (dia==7){
  console.log("Julio");
}
```

```
else if (dia==8){
  console.log("Agosto");
}
else if (dia==9){
  console.log("Septiembre");
}
else if (dia==10){
  console.log("Octubre");
}
else if (dia==11){
  console.log("Noviembre");
}
else{
  console.log("Diciembre");
}
```

Si lo pasamos a estructura switch, quedaría:

```
var dia = parseInt(prompt("Número de mes?", ""));

//Dependiendo del dia que sea, hago un saludo u otro

switch (dia){
  case 1:
    console.log("Enero");
    break;
  case 2:
    console.log("Febrero");
    break;
  case 3:
    console.log("Marzo");
    break;
```

```
  case 4:
    console.log("Abril");
    break;
  case 5:
    console.log("Mayo");
    break;
  case 6:
    console.log("Junio");
    break;
  case 7:
    console.log("Julio");
    break;
  case 8:
    console.log("Agosto");
    break;
  case 9:
    console.log("Septiembre");
    break;
  case 10:
    console.log("Octubre");
    break;
  case 11:
    console.log("Noviembre");
    break;
  default:
    console.log("Diciembre");
}
```

Bucles 8

Hasta hace un par de capítulos, nuestros scripts han sido tremendamente sencillos y lineales: se iban ejecutando las sentencias simples una detrás de la otra desde el principio hasta el fin.

En el capítulo anterior de este manual añadimos la capacidad de crear "caminos de ejecución" dependiendo de si se cumplían o no las condiciones que indicábamos.

Ahora es el turno de los bucles. Los bucles se utilizan para repetir partes de código un numero determinado de veces. Con un bucle podemos por ejemplo imprimir en una página el mismo mensaje 100 veces sin necesidad de escribir cien veces el la instrucción imprimir. La idea es la siguiente:

Desde el 1 hasta el 100
  Imprimir el mensaje

Al igual que en otros lenguajes de programación, En javascript existen varios tipos de bucles:

- FOR
- WHILE
- DO WHILE

8.1 Bucle FOR

El bucle FOR se utiliza para repetir instrucciones un determinado número de veces. De entre todos los bucles, el FOR se suele utilizar cuando sabemos seguro el número de veces que queremos que se ejecute la sentencia. La sintaxis del bucle se muestra a continuación:

```
for (i=valor inicial;condición;actualización) {
   sentencias a ejecutar en cada vuelta
}
```

El bucle FOR tiene una cabecera compuesta por tres partes incluidas entre paréntesis.
La primera es el valor inicial, que se ejecuta solamente al comenzar la primera iteración del bucle. En esta parte se suele colocar una variable definida por nosotros que utilizaremos para llevar la cuenta de las veces que se ejecuta el bucle (Para nosotros será la variable '`i`').

La segunda parte es la condición, que se evaluará cada vez que de una vuelta el bucle. Esta condición sirve para decidir cuando se para el bucle: mientras la condición se cumpla, el bucle seguirá repitiendo código.

Por último tenemos la actualización, que sin entrar mucho en detalle, para nosotros siempre será la sentencia `i++`

Después de la cabecera explicada, se colocan entre llaves las sentencias que queremos que se repitan cada vez que de vueltas el bucle.

Un **ejemplo** de utilización de este bucle: vamos a imprimir 11 veces la frase "*Ya entiendo los bucles*"

```
var i;
for (i=0;i<=10;i++) {
  console.log("Ya entiendo los bucles");
}
```

Importante darse cuenta de que en la cabecera se han definido 11 del 0 al 10. Podríamos haber puesto del 1 al 11 o del 5 al 16.

En este caso se inicializa la variable `i` a 0. Como condición para realizar una vuelta, se tiene que cumplir que la variable `i` sea menor o igual que 10.Al final de cada vuelta se incrementará en 1 la variable `i`, por lo que nos aseguramos que en algún momento el bucle va a parar.

Si quisiéramos escribir los número del 1 al 1.000 se escribirá el siguiente bucle.

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

Lo primero que hacemos es indicar el numero de vueltas que debe dar el bucle: desde 1 hasta 1000. A continuación, en cada vuelta mostramos el valor de `i` en cada momento: en la primera vuelta vale 1, en la segundo2, en la tercera 3… esto funciona porque con `i++` indicamos que se incremente el valor de `i` en uno tras cada vuelta.

Es posible omitir las llaves que engloban las instrucciones del bucle FOR. Esto ocurre cuando sólo hay una sentencia a repetir dentro del bucle. En este caso, no es obligatorio el uso de las llaves.

Aun así, se recomienda poner las llaves siempre para evitar confusiones dado que si no ponemos las llaves y escribimos mas de una sentencia dentro del bucle, el FOR repetirá solo la primera sentencia que encuentre, el resto solo se ejecutará una vez.

```
for (i=1;i<=1000;i++)
   console.log("Repetido");
   console.log("hola");
```

En el ejemplo anterior la palabra *repetido* aparecerá 1000 veces en consola (lo que indica el bucle FOR) sin embargo, la palabra *hola* tan solo aparecerá una vez dado que hemos omitido las llaves. Fijate que la tabulación no influye en cómo deben ejecutarse las sentencias.

8.1.1 Ejemplo

Vamos a hacer una pausa para asimilar el bucle FOR con un ejercicio que no encierra ninguna dificultad si hemos entendido el funcionamiento del bucle.

Se trata de hacer un bucle que escriba en una página web los encabezamientos desde `<h1>` hasta `<h6>` con un texto que ponga "*Encabezado de nivel x*".

Lo que deseamos escribir por consola mediante Javascript es lo siguiente:

```
<H1>Encabezado de nivel 1</H1>
<H2>Encabezado de nivel 2</H2>
<H3>Encabezado de nivel 3</H3>
<H4>Encabezado de nivel 4</H4>
<H5>Encabezado de nivel 5</H5>
<H6>Encabezado de nivel 6</H6>
```

Para ello tenemos que hacer un bucle que empiece en 1 y termine en 6 y en cada vuelta escribiremos el encabezado que toca utilizando la función `console.log` de Javascript.

```
for (i=1;i<=6;i++) {
   console.log("<h"+i+">Encabezado de nivel "+i+"</h"+i+">");
}
```

```
<h1>Encabezado de nivel 1</h1>
<h2>Encabezado de nivel 2</h2>
<h3>Encabezado de nivel 3</h3>
<h4>Encabezado de nivel 4</h4>
<h5>Encabezado de nivel 5</h5>
<h6>Encabezado de nivel 6</h6>
```

8.1.2 Guion de funcionamiento del bucle FOR

En los apartados anteriores, hemos dado muy por encima el funcionamiento del bucle FOR. Hay que indicar que este bucle es muy potente y existen más formas de usarlo. Sin embargo, para el nivel que pretende obtener este manual, para nosotros los bucles FOR siempre van a tener la misma forma. De este modo, podemos describir su funcionamiento en unos pequeños pasos:

1. Se crea la variable '`i`' y se llena con el valor inicial.
2. Se realiza la pregunta: es '`i`' menor (o igual) que …
3. En caso de que se cumpla la condición ejecuto las sentencias que hay dentro de las llaves del bucle FOR una a una.
4. Incremento en una unidad la variable '`i`'
5. Vuelvo al paso 2.
6. En caso de que no se cumpla la condición, salgo del bucle (voy a la llave de cierre) y ACABO. Ya no doy mas vueltas.

Hay que destacar dos cosas muy importantes que suceden en los bucles FOR:

- Lo que va entre sus llaves se repite un número determinado de veces.
- La variable '`i`' va a ir cambiando de valor conforme de vueltas el bucle.

8.1.3 Otro ejemplo

Como ya hemos comentado, el bucle FOR tiene '*mas chicha*' que la que hemos explicado aquí. Un ejemplo de la versatilidad de este bucle podría ser el contar descendentemente.

Vamos a mostrar un bucle que vaya desde el 343 hasta el 10 (343 – 10 = 333 vueltas) de forma descendente:

```
for (i=343;i>=10;i--){
 console.log(i);
}
```

En este caso, tras cada vuelta loq ue hacemos es decrementar en una unidad la variable `i`. Para ello usamos el `i--` definido en al cabecera.

A tener en cuenta: Siempre que defina el número de vueltas desde un número menor a otro mayor debo INCREMENTAR el valor de al variable `i`. Así mismo, defina el número de vueltas desde un numero mayor a otro menor, debo DECREMENTAR el valor de al variable `i`.

8.2 Bucle WHILE

Este bucles se utiliza cuando queremos repetir la ejecución de unas sentencias un número indefinido de veces (no sabemos las vueltas que tenemos que dar). El bucle siempre va a estar dando vueltas mientras que se cumpla una condición.

Es más sencillo de comprender que el bucle FOR, puesto que sólo hay que indicar, la condición que se tiene que cumplir para que se realice una vuelta.

```
while (condición){
    sentencias a ejecutar
}
```

Un ejemplo de código donde se utiliza este bucle se puede ver a continuación.

```
var color;
color = prompt("Dame un color","");

while (color != "rojo"){
  color = prompt("Dame un color","");
}
console.log("has elegido el color rojo");
```

Este es un ejemplo de lo más sencillo que se puede hacer con un bucle while:

- Lo que hace es pedir que el usuario introduzca un color.
- Mientras que el color no sea rojo, se vuelve a pedir al usuario que introduzca un color.
- Cuando el color sea `"rojo"`, no se cumple la condición del `while` y, por tanto, dejo de dar vueltas.

- Al salir del bucle se sigue ejecutando el programa y se imprime en la consola el mensaje *has elegido el color rojo*. [8]

Hay que tener **mucho cuidado** con este bucle: Siempre tenemos que estar seguro que la condición del bucle va a dejar de cumplirse en algún momento, si no, lo que metamos dentro del bucle `while` se estará repitiendo infinitas veces y así conseguiremos dejar colgado el navegador.

Por ejemplo:

```
var num=7;

while (num < 10){
  alert("Soy un pesado");
}
```

En la primera linea, la variable `num` vale 7. La condición del `while` dice que *"mientras num sea menor que 10*" doy vueltas y muestro el `alert`. Como no hay nada que haga cambiar el valor de `num`, esa variable siempre será 7 y, por tanto, siempre será menor que 10. Así que,el bucle `while` no para nunca y se produce loq ue en programación se llama **bucle infinito**.

8.3 Bucle DO..WHILE

Es el último de los bucles que hay en Javascript. Es exactamente igual que el bucle WHILE con la diferencia de que sabemos seguro que el bucle **por lo menos se ejecutará una vez** (recordamos que en el WHILE, si no se cumple la condición al principio, no entramos en el bucle).

La sintaxis es la siguiente.

```
do {
   sentencias del bucle
} while (condición)
```

8 Fíjate que esta frase sólo va a poder mostrarse cuando el bucle pare, nunca antes de eso.

El bucle se ejecuta siempre una vez y al final se evalúa la condición para decir si se ejecuta otra vez el bucle o se termina su ejecución.

El ejemplo que vimos en el bucle WHILE pedía al usuario un color hasta que fuera 'rojo'.

```
var color;
color = prompt("Dame un color","");

while (color != "rojo"){
  color = prompt("Dame un color","");
}
console.log("has elegido el color rojo");
```

Fijate que primero pregunta por el color y luego mira si entra o no en el bucle. Si entra en el bucle, deberá seguir preguntando. De ahí que se necesiten dos sentencias `prompt`.

Si hacemos este mismo ejemplo usando el bucle DO..WHILE nos quedaría así:

```
var color;
do {
  color = prompt("Dame un color","");
}while (color != "rojo");
console.log("has elegido el color rojo");
```

Como el bucle se ejecuta al menos una vez, ahora solo es necesario escribir una sentencia `prompt` puesto que en este caso ya estamos dentro del bucle, no hay que decidir nada para entrar (la condición decide si se siguen dando vueltas o no).

8.4 La sentencia BREAK

La sentencian `break` se puede usar dentro de muchas de las estructuras y bucles que hemos visto en este tema. Ya explicamos su uso dentro de la estructura SWITCH.

Sin embargo, si usamos esta sentencia dentro de un bucle, `break` lo que se consigue es parar forzosamente las vueltas de ese bucle. Esto significa que se sale de él y deja todo como está en esa vuelta (valores de i, condiciones...) queden las vueltas que queden por dar.

Como es lógico, Al salir del bucle, el flujo de ejecución continua después del bucle.

```
for (i=1;i<=1000;i++){
    console.log(i);
    if (i==25){
       break;
    }//if
}
```

En el código anterior vemos que se trata de un bucle FOR planeado para dar 1.000 vueltas (desde 1 hasta 1000) pero que se para forzosamente con un con `break` cuando se llega la valor 25.

En el siguiente ejemplo escribe los números del 1 al 10 y en cada vuelta del bucle pregunta al usuario si desea continuar. Si escribe "no" que entonces se sale del bucle y deja la cuenta por donde se había quedado.

```
for (i=1;i<=10;i++){
    console.log(i);
    escribe = prompt("continuo? (si/no)","");
    if (escribe == "no"){
     break;
    }//if
}
```

Importante: El uso de la sentencia `break` no es una buena practica de programación debido al abuso que se realiza de ella. En la mayoría de los casos, se puede evitar su uso rompiendo la condición del bucle, por lo que, si se usa para salir de un bucle, es señal de que el programador que lo hace no tiene unas buenas bases del funcionamiento de los bucles en programación.

```
for (i=1;i<=1000;i++){
    console.log(i);
    if (i==25){
       i = 1001; //pongo un valor que impida continuar
    }//if
}
```

8.5 Continue

La sentencia `continue` se usa en bucles y sirve para volver al principio del bucle en cualquier momento, sin ejecutar las líneas que haya por debajo de la palabra `continue` (es decir, se usa para 'saltarse' código en las vueltas de un bucle).

```
var i, inc;

while (i<7){
  inc = prompt("sumo uno a la variable? (si/no)","");
   if (inc == "no"){
      continue; //me salto todo desde desde este punto
                //hasta la llave de cierre del bucle
   }
   i++;
   console.log("La variable se ha incrementado");
}
```

Este ejemplo, en condiciones normales contaría hasta desde `i=0` hasta `i=7`, pero cada vez que se ejecuta el bucle pregunta al usuario si desea incrementar la variable o no. Si introduce "no" se ejecuta la sentencia `continue,` con lo que se vuelve al principio del bucle sin llegar a incrementar en 1 la variable `i` ni a mostrar el mensaje en la consola, ya que se ignoran las sentencia que hayan por debajo del `continue` (se salta todo lo que queda en esa vuelta).

8.6 Bucles anidados

Anidar un bucle consiste en meter ese bucle dentro de otro. La anidación de bucles es algo que se usa normalmente en programación y que permite hacer determinadas acciones más complejas de las que se han visto hasta el momento

El ejemplo típico de un bucle FOR anidado tiene una estructura parecida a la que se muestra a continuación:

```
for (i=0;i<=3;i++){
    for (j=0;j<=9;j++) {
       console.log(i + "-" + j)
    }
}
```

La ejecución funcionará de la siguiente manera:

- Para empezar se inicializa el primer bucle, con lo que la variable i valdrá 0
- A continuación se inicializa el segundo bucle, con lo que la variable j valdrá también 0.
- En cada iteración se imprime el valor de la variable i, un guión ("-") y el valor de la variable j,
- En la primera vuelta, las dos variables valen 0 por lo que se imprimirá el texto "0-0" en consola.
- Ahora es el bucle de dentro (`j`) el que debe dar todas sus vueltas primero. Por lo que la `j` valdrá 1,2,… hasta dar 10 vueltas
- En la página consola se mostrarán los valores:

```
0-0
0-1
0-2
0-3
0-4
0-5
0-6
0-7
0-8
0-9
```

- Una vez acabe el bucle interno (`j`), el bucle externo (`i`) dará otra vuelta. Y el bucle interno volverá a dar sus 10 vueltas correspondientes. Es decir, para cada vuelta del bucle externo (`i`) se ejecutarán las 10 vueltas del bucle interno o anidado (`j`).

- Así pues se van a obtener los valores:

```
1-0
1-1
1-2
1-3
1-4
1-5
1-6
1-7
1-8
1-9
```

Y luego estos:

```
2-0
2-1
2-2
2-3
2-4
2-5
2-6
2-7
2-8
2-9
```

y finalmente:

```
3-0
3-1
3-2
3-3
3-4
3-5
3-6
3-7
3-8
3-9
```

Veamos **otro ejemplo** muy parecido al anterior, aunque un poco más útil. Se trata de imprimir en la página las todas las tablas de multiplicar. Del 1 al 9, es decir, la tabla del 1, la del 2, del 3...

```
for (i=1;i<9;i++){
    console.log("La tabla del " + i + ":");
    for (j=1;j<9;j++) {
      console.log(i + " x " + j + "="+(i*j));
    }
}
```

Con el primer bucle controlamos la tabla actual y con el segundo bucle la desarrollamos. En el primer bucle escribimos una cabecera indicando la tabla que estamos escribiendo, primero la del 1 y luego las demás en orden ascendente hasta el 9.

Con el segundo bucle escribo cada uno de los valores de cada tabla:

```
La tabla del 1:
1x1=1
1x2=2
1x3=3
1x4=4
1x5=5
1x6=6
1x7=7
1x8=8
1x9=9

La tabla del 2:
2x1=2
2x2=4
2x3=6
2x4=8
2x5=10
2x6=12
2x7=14
2x8=16
2x9=18

...
```

Arrays 9

En los lenguajes de programación existen estructuras de datos especiales que nos sirven para guardar información más compleja que la que podemos almacenar en simples variables.

La estructura más típica en todos los lenguajes de programación es el Array. Un array es "una variable" en la que vamos a poder introducir una colección de valores, en lugar de solamente un valor como ocurre con las variables normales.

Un array se asemeja a una cajonera (un mueble con varios cajones). Ese mueble es todo un bloque, el cual tiene varios cajones donde podemos meter y sacar cosas.

El array sería ese mueble y cada "cajón" del array funcionaría exactamente igual que una variable.

9.1 Definición de Arrays

El primer paso para utilizar un array es definirlo o crearlo. La sentencia para crear un array en JavaScript es:

```
var mueble = new Array(5)
```

Con esto se crea un array llamado `mueble` el cual va a tener 5 cajones. Cada cajón del array va a estar numerado desde el 0 hasta el tamaño indicado, es decir. como en la mayoría de lenguajes de programación, en Javascript los cajones de un array empiezan a numerarse desde el cero.

Sabiendo esto, el array `mueble` va a tener cinco cajones que son: el 0, el 1,el 2,el 3 y el 4.

Es importante que nos fijemos que la palabra Array en código Javascript se escribe con la primera letra en mayúscula. Como en Javascript las mayúsculas y minúsculas si que importan, si lo escribimos en minúscula no funcionará.

Como hemos comentado, cada cajón de un array funciona exactamente igual que una variable. Es decir, voy a poder meter datos y/o consultar lo que tiene.

Para usar un cajón del array es necesario referenciarlo por su número.

Así pues, para introducir valores en los cajones de un array, tan solo hay que indicar que cajón es el que voy a "abrir" y meter en él lo que queramos (como ya hemos visto en el tema de variables):

```
mueble[0] = 290
mueble[2] = 127
```

Aquí estoy metiendo el numero 290 en el primer cajón del array (cajón 0) y el numero 127 en el tercer cajón del array (cajón 2).

Como vemos, se debe indicar el nombre del array y, entre corchetes, el índice del cajón donde queríamos guardar el dato.

Para sacar datos de un array lo hacemos igual: poniendo entre corchetes el índice del cajón que queremos abrir.

Ejemplo: para mostrar por consola el contenido de un array podemos hacer lo siguiente:

```
let miArray = new Array(3)

miArray[0] = 155
miArray[1] = 4
miArray[2] = 499

console.log(miArray[0]);
console.log(miArray[1]);
console.log(miArray[2]);
```

Importante: Hoy día , en pocas ocasiones se van a definir arrays en Javascript como se ha comentado en el punto anterior. No porque esté mal sino porque existe una forma mucho más rápida de hacerlo: La notación JSON.

Esta "forma de definir estructuras complejas" en Javascript se verá con detenimiento en la parte Avanzada de este manual.

9.2 Tipos de datos en arrays

Como ya hemos dicho, los cajones de un array se comporta exactamente igual que las variables. Por tanto, en las casillas de los arrays podemos guardar datos de cualquier tipo de los que conocemos.

Pej, podemos tener un array donde introducimos datos de tipo cadena.

```
miArray[0] = "Hola"
miArray[1] = "a"
miArray[2] = "todos"
```

A diferencia de lenguajes fuertemente tipados (como JAVA o C++), en *Javascript* podemos guardar distintos tipos de datos en los cajones de un mismo array. Es decir, podemos introducir números en unos cajones, textos en otros, booleanos o cualquier otra cosa que deseemos.

```
miArray[0] = "Que interesante es la clase"
miArray[1] = 1275
miArray[2] = 0.78
miArray[3] = true
```

9.3 Longitud de un array: length

Todos los arrays en javascript, aparte de almacenar el valor de cada una de sus posiciones también almacenan el número de cajones totales que tienen. Para ello utilizan una propiedad del array, la propiedad `length`.

En este punto del manual, es suficiente con entender que `length` es una variable que se crea automáticamente al crear el array y que almacena un número igual al número de cajones totales del array.

Esta variable solo puede consultarse (no puede cambiarse). Para ello se escribe el nombre del array del que queremos saber el número de posiciones que tiene, sin corchetes ni paréntesis, seguido de un punto y la palabra `length`.

```
var miArray = new Array(3)

miArray[0] = 155
miArray[1] = 499
miArray[2] = 65

console.log("Longitud del array: " + miArray.length);
```

Este código imprimiría en pantalla el número de posiciones del array, que en este caso es 3 (que es el mismo número que hemos dado al crear el array). Recordamos que un array con 3 posiciones abarca desde la posición 0 a la 2.

Aunque no lo parezca, esto tiene su utilidad. Piensa que la mayoría de las veces no vamos a saber el numero de cajones que tiene un array (mas que nada porque no lo habremos creado nosotros). Pej, un array con todos los usuarios registrados en tu pagina.

9.4 Inicialización de arrays

Al igual que las variables, los arrays tienen una manera de inicializar sus valores a la vez que lo declaramos, así podemos realizar de una forma más rápida el proceso de introducir valores en cada una de las posiciones del array.

El método normal de crear un array vimos que era a través del objeto Array, poniendo entre paréntesis el número de casillas del array. Para introducir valores a un array se hace igual, pero poniendo entre los paréntesis los valores con los que deseamos rellenar las casillas separados por coma.

Veámoslo con un ejemplo que crea un array con los nombres de los días de la semana.

```
var diasSemana = new Array ("Lunes", "Martes", "Miércoles",
"Jueves",     "Viernes",     "Sábado",     "Domingo")
```

El array se crea con 7 casillas, de la 0 a la 6 y en cada casilla se escribe el día de la semana correspondiente (Entre comillas porque es un texto).

Es obvio que podemos rellenar los cajones del array con valores de los diferentes tipos que conocemos (números, cadenas de textos y booleanos):

```
var mezcla = new Array ("Lunes", 3.14159, "true", false,
23, "11");
```

9.5 Control de arrays con el bucle FOR

El uso del bucle FOR parra recorrer arrays es de lo mas habitual en programación. Esto es porque un bucle FOR da siempre el número de vueltas que le indiquemos. Vueltas que se pueden aprovechar para meter/sacar cosas en los cajones de un array de forma rápida y limpia (escribiendo unas pocas lineas de código).

Para ello basta con tener en cuenta que el primer cajón de un array es el 0 y el último es el número total de cajones menos 1.

Vamos a suponer un array que tiene 12 cajones y que en cada cajón hemos metido el nombre de uno de los meses del año:

```
var  meses = new Array (12).


meses[0] = "Enero";
meses[1] = "Febrero";
meses[2] = "Marzo";
…
meses[11] = "Diciembre";
```

Con lo aprendido hasta ahora, si ahora quisiéramos mostrar todos los cajones, nos quedaría mas remedio que hacer algo como esto:

```
console.log(meses[0]);
console.log(meses[1]);
console.log(meses[2]);
…
console.log(meses[11]);
```

Habría que escribir mucho. Y aún seria peor si nuestro array tuviera miles de cajones (cosa frecuente en el tema web junto con Bases de Datos).

Para recorrer un array de forma rápida se usa el bucle FOR:

- Indicamos las vueltas que vamos a dar (desde 0 hasta numero total d ecajones menos 1).
- Para cada vuelta, la variable 'i' va a tomar cada uno d ellos números correspondientes a los cajones del array. De esta forma, podemos usar esa variable para abrir un cajón distinto en cada vuelta.

Para mostrar el contenido del array meses nos basta con hacer:

```
for(let i=0;i<=11;i++){ //empiezo en 0 y acabo en 11 (12-1)
  console.log(meses[i]);
}
```

Se nota la diferencia entre un código y otro. Ahora hemos usado una sola línea. Da igual que el array tenga 1 cajón que tenga 1000. Con esa línea (y los valores correctos en el FOR) va a mostrar todo el array.

Esto es gracias a que la variable `i` del FOR se va a mover entre 0 y 11 (o el valor que le digamos). Nosotros usamos la `i` para que, dependiendo de la vuelta en la que esté, abra un cajón u otro.

También, es muy habitual que se utilice la propiedad `length` (vista en el tema anterior) para poder recorrer un array, del cual no conocemos su tamaño, por todas sus posiciones. Para ilustrarlo vamos a ver un ejemplo de recorrido por este array para mostrar sus valores.

```
for(let i=0;i<=(miArray.length -1);i++){
  console.log(miArray[i]);
}
```

Echo esto, el bucle FOR se ejecutará siempre que ‘`i`’ valga menos que la longitud del array, extraída de su propiedad `length`.

9.6 Dinamismo en los arrays

A diferencia de otros lenguajes, En Javascript los arrays son dinámicos, es decir, si intentamos acceder a un cajón que no existe, este cajón se crea automáticamente, así como todos los que "falten" hasta llegar a él. Además, el valor de la propiedad `length` queda actualizado también.

El siguiente ejemplo nos servirá para entender lo que se acaba de comentar: Vamos a crear un array con 2 posiciones y a rellenalos. Posteriormente introduciremos un valor en la posición 5 del array (posición que no existe por que hemos definido solo 2 cajones). Finalmente imprimiremos todas las posiciones del array para ver lo que pasa.

```
var miArray = new Array(2)

miArray[0] = "Colombia";
miArray[1] = "Estados Unidos" ;

miArray[5] = "Brasil";

for (let i=0;i<miArray.length;i++){
 console.log("Posición " + i + " del array: " + miArray[i]);
}
```

El ejemplo es sencillo. Se puede apreciar que hacemos un recorrido por el array desde 0 hasta el número de posiciones del array (indicado por la propiedad `length`). En el recorrido vamos imprimiendo el número de la posición seguido del contenido del array en esa posición. Pero podemos tener una duda al preguntarnos cuál será el número de elementos de este array, ya que lo habíamos declarado con 2 y luego le hemos introducido un tercero en la posición 5. Al ver la salida del programa podremos contestar nuestras preguntas. Será algo parecido a esto:

```
Posición 0 del array: Colombia
Posición 1 del array: Estados Unidos
Posición 2 del array: undefined (o empty)
Posición 3 del array: undefined (o empty)
Posición 4 del array: undefined (o empty)
Posición 5 del array: Brasil
```

Se puede ver claramente que el número de posiciones es 6, de la 0 a la 5. Lo que ha ocurrido es que al introducir un dato en la posición 5, todas las casillas que no estaban creadas hasta la quinta se crean también.

Las posiciones de la 2 a la 4 están sin inicializar. En este caso nuestro navegador ha escrito la palabra *null* para expresar esto, pero otros navegadores podrán utilizar la palabra *undefined.*

9.7 Usando WHILE para recorrer arrays

Dado que `while` es el mas poderoso de los bucles en programación y todo lo que hagan el resto de bucles, se puede hacer con `while`. Es obvio que, si con FOR hemos podido recorrer arrays, con while también es posible.

Como ya vimos, usando un FOR sería así:

```
var lista = new Array(5);


//relleno el array
…
//muestro su contenido
for(let i=0;i<=4;i++){
   console.log(lista[i]);
}
```

Con un `while` también es sencillo. Tan solo hay que fijarse en que esta vez, somos nosotros los que tenemos que crear la variable `i`, ver si la variable es menor que el tope de vueltas que queremos dar (en este ejemplo, 4) e incrementar su valor.

```
var lista = new Array(5);


//relleno el array
…
//muestro su contenido
let i = 0;
```

```
while (i<=4){
   document.write(lista[i]);
   i++;
}
```

Destacar que, lo que está puesto en negrita, es justo lo mismo que hay dentro del FOR (y en el mismo orden). Asi que, en el caso de que el lector no sea capaz de razonar la equivalencia entre el FOR y el WHILE, siempre puede recurrir al truco de colocar los trozos del FOR en el WHILE como se ha indicado en el ejemplo anterior.

Es importante indicar que el bucle `while` se usa con arrays cuando se realizan búsquedas de valores en el array: sigo en el bucle mientras no lo encuentre. Si lo encuentro, acabo el bucle (aunque no haya acabado el array).

Con un bucle FOR recorremos todo el array, por tanto, usar un bucle FOR para buscar elementos en el array es un fallo gordo de programación.