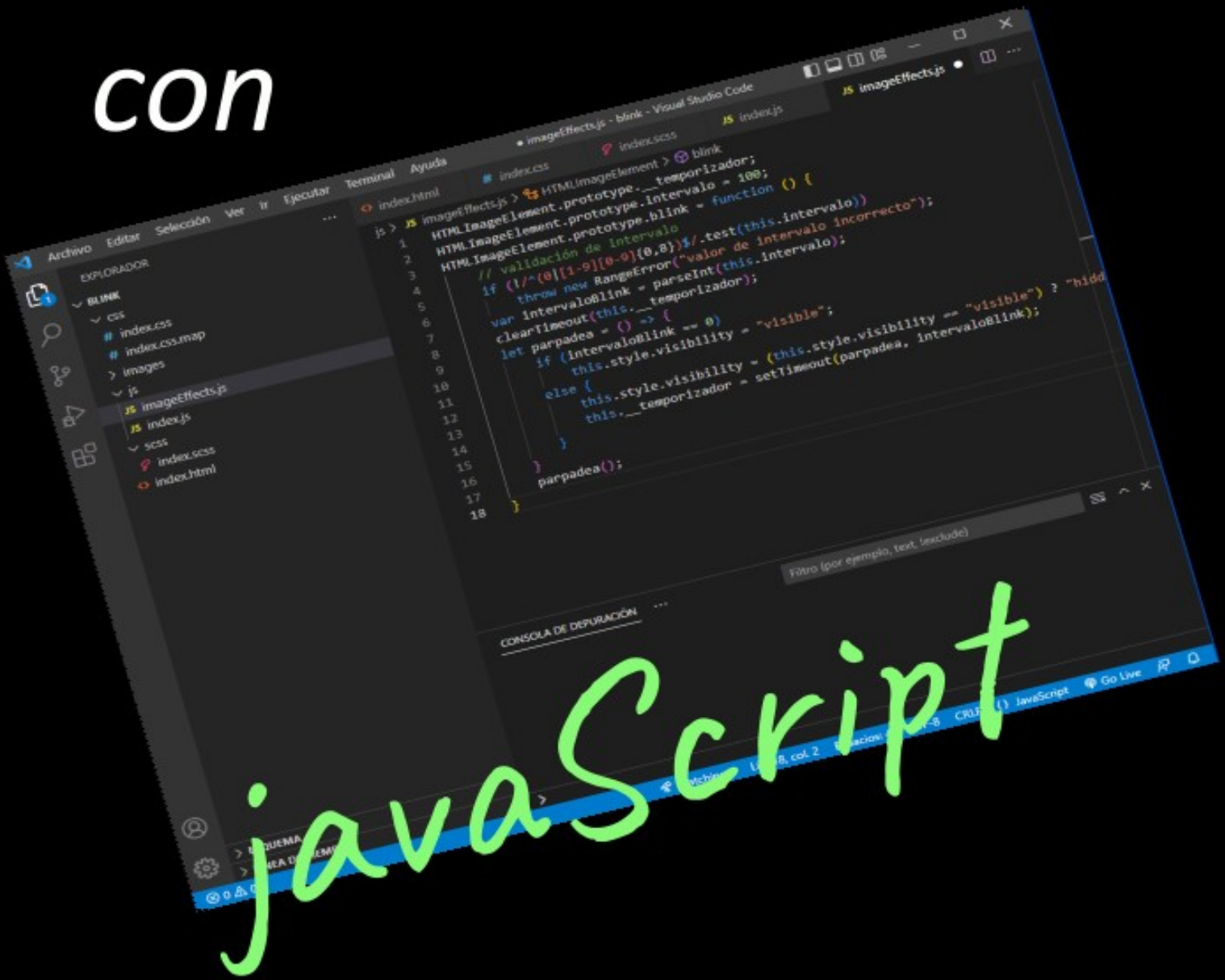
Iniciación a la programación con



Indice

Iniciación a la programación con javaScript

Introducción

Un **ordenador**, o en un concepto más general un **procesador**, no es más que una máquina electrónica capaz de transformar mediante procesos lógicos y aritméticos una información de entrada suministrada en forma de datos, dando lugar a unos nuevos datos y por tanto a una nueva información. Esta nueva información puede, por ejemplo, desencadenar acciones sobre otros componentes electrónicos, o servir de información de entrada a nuevos procesos … Por ejemplo, un valor de temperatura tomado desde un sensor en un secadero de jamones puede desencadenar la puesta en marcha de un sistema de refrigeración, si es que este nuevo valor supera un límite establecido.

La forma en que la información que entra es transformada, viene dada por una serie de instrucciones dadas al procesador. Estas instrucciones es a lo que se denomina **programa**. Los procesadores son capaces de realizar operaciones básicas tanto aritméticas como lógicas a una gran velocidad y sin cometer errores. Por ser dispositivos electrónicos basados en tecnologías de dos estados, tanto el manejo de los datos como su programación se hace codificándolos en formato binario.

Los seres humanos, por otro lado, utilizamos el **lenguaje natural**, los idiomas, para comunicarnos entre nosotros por lo que el código binario para representar la información y los datos se nos hace muy extraño. Aquí nace el concepto de **lenguaje de programación** y de los **códigos de representación de información**. Un lenguaje de programación es un **lenguaje**, una serie de palabras ordenadas de acuerdo a una reglas de sintaxis, más o menos cercano al lenguaje natural humano, que representan instrucciones para ser utilizadas en la programación de un procesador. Está claro que si el procesador solo entiende de código binario, y el lenguaje de programación está escrito en un código cercano al lenguaje natural humano debe haber un proceso por el cual se traduzca este a código binario entendible por el procesador. Este proceso de traducción es el que es realizado por los **compiladores e intérpretes** de lenguajes de programación.

La principal diferencia entre compilador e intérprete es la forma en que se realiza ese proceso de traducción.

El **compilador** traduce traduce todo el código escrito en un determinado lenguaje de programación, **código fuente**, a código binario, **código máquina**, generando un archivo con el código ya traducido. Una vez traducido ya no habrá que volver a traducir el código fuente cuando se deseen ejecutar las instrucciones. Un compilador detectará todos los errores de sintaxis del código fuente en esta fase de traducción y hasta que el código fuente no esté libre de errores no se tendrá el código ejecutable traducido.

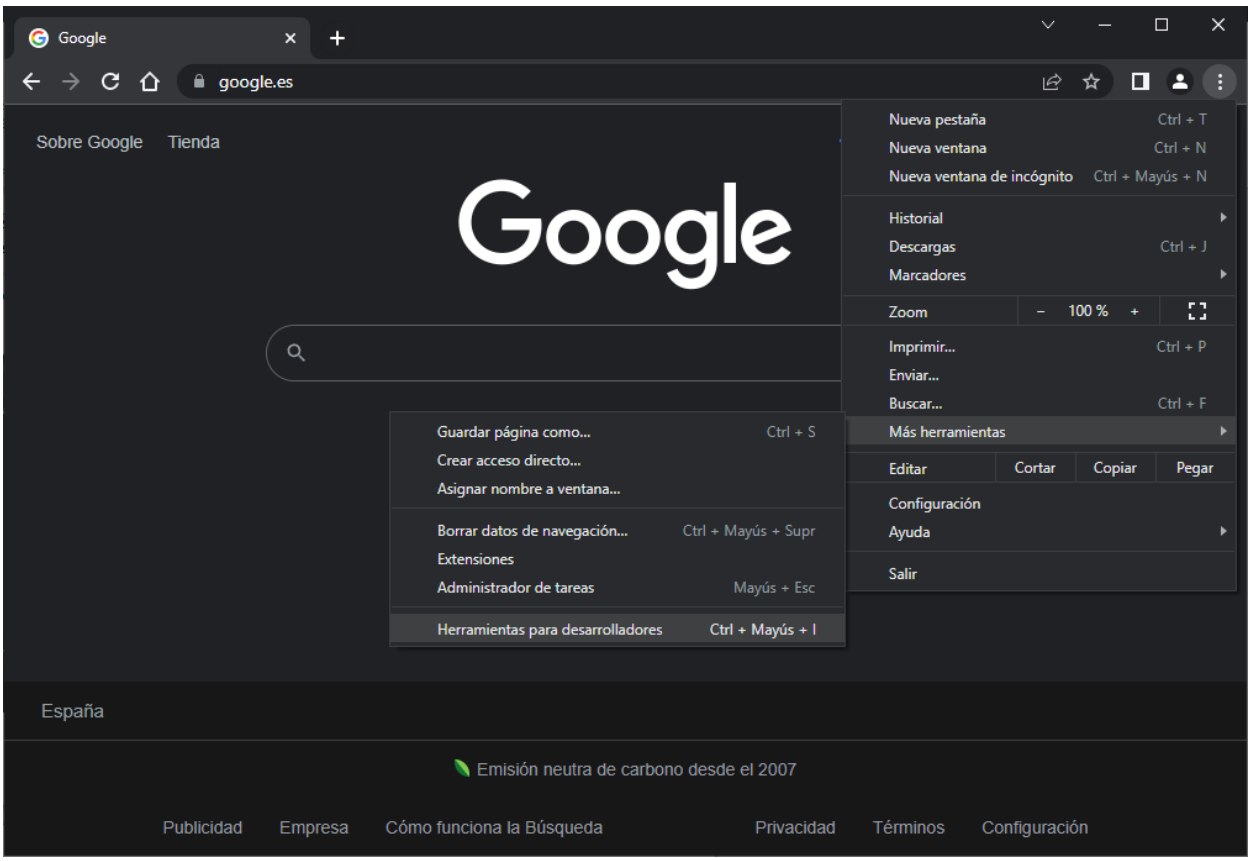
Por contra, un **intérprete** solo traducirá la siguiente instrucción de código fuente que vaya a ser ejecutada, por lo tanto, múltiples ejecuciones del código darán lugar a múltiples traducciones. Se deduce también que el intérprete no detectará errores de sintaxis hasta que la instrucción que los contenga no vaya a ser ejecutada.

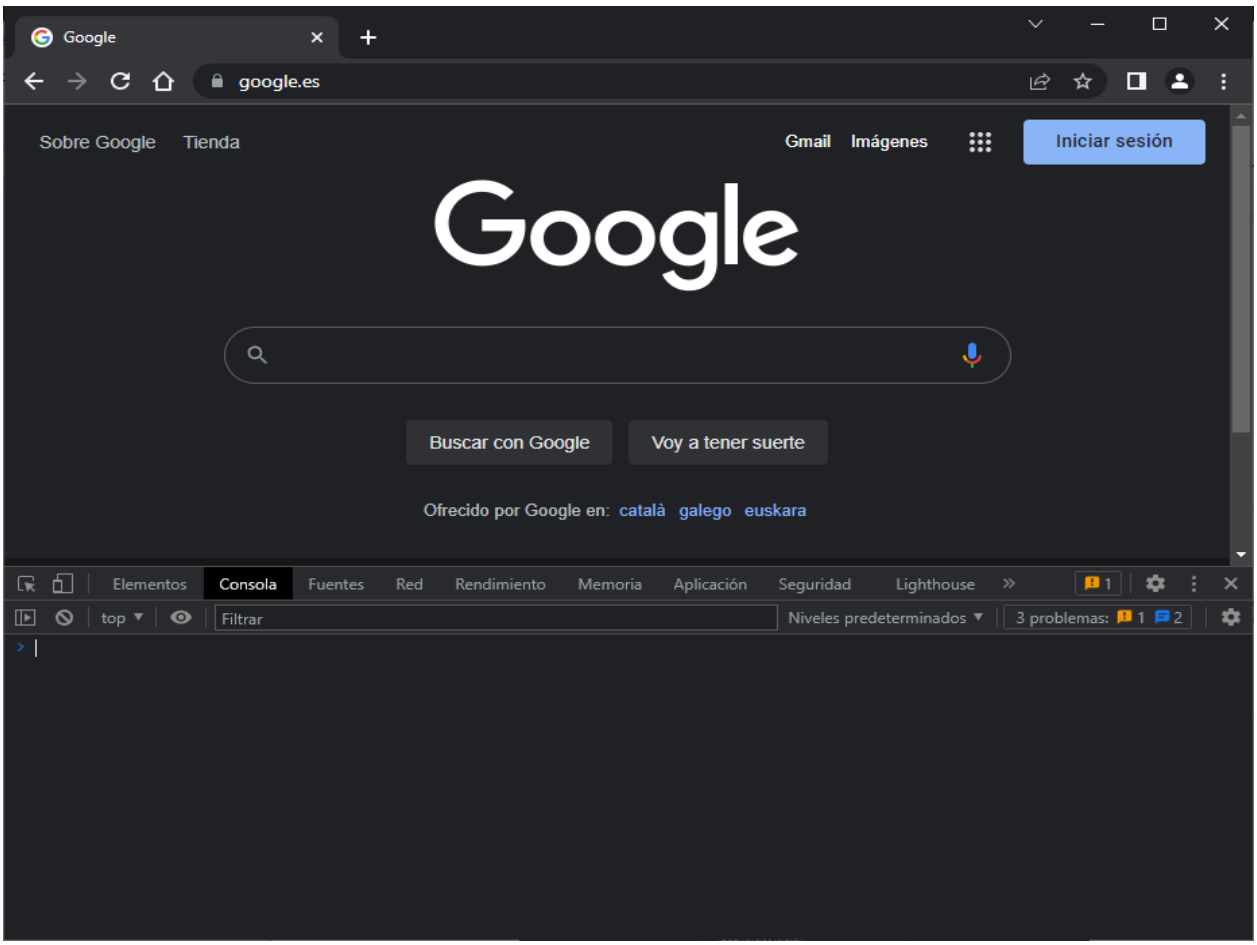
En esta **iniciación a la programación** vamos a utilizar el lenguaje **javaScript** que es un lenguaje interpretado y por lo tanto, cada vez que queramos ejecutar el código escrito en este lenguaje tendremos que traducir el código, de lo que se encargará el intérprete. Podemos encontrar, por ejemplo, intérprete en el que incorporan los navegadores web para añadir funcionalidad a las páginas web, o en el sistema operativo Windows para ejecutar scripts (programas de comandos del sistema). Vamos a aprender javaScript haciendo uso del intérprete que incorporan los navegadores web, no necesitando instalar ninguna utilidad nueva relacionada con el lenguaje.

El código fuente tendremos que escribirlo haciendo uso de un **editor de texto**. Un editor de texto es un programa que incorporan todos los sistemas operativos y su función, como su nombre indica, es permitir al usuario la *edición* de texto, es decir, su manipulación: añadir, modificar, borrar, guardar en archivo, etc. sin añadir elementos extraños como hace, por ejemplo, un **procesador de texto** que añade marcas internas invisibles para permitir, por ejemplo, poner texto en negrita o en distintos tamaños. **Notepad** es un editor de texto y **Word** es un procesador de texto. Existen programas especializados en la edición de texto para escribir código fuente en distintos lenguajes de programación como por ejemplo **notepad++**. Si el programa editor de texto además ofrece la posibilidad de lanzar la ejecución del traductor o compilador y controlar la ejecución del código, **debug,** estaremos hablando de un **IDE**, por ejemplo, **Visual Studio** de **Microsoft** que es un entorno de programación integrado (**I**ntegrated **D**evelopment **E**nvironment) para los lenguajes de programación de **Microsoft**. Hay editores de texto que permiten lanzar la compilación y ejecución, así como la depuración (control de la ejecución) enlazando con herramientas externas, siendo compatibles con multitud de lenguajes de compilación. Es el caso de **Eclipse** o el que se va a utilizar en esta introducción que es el editor **Visual Studio Code** de **Microsoft** que es un editor de texto multilenguaje, multiplataforma y gratuito que tiene todas estas funcionalidades mencionadas. Se puede descargar desde **https://code.visualstudio.com/download**

Primeros pasos

Vamos a realizar nuestro primer programa y por ahora no vamos a utilizar un editor separado para codificarlo y ejecutarlo. Nuestro primer programa es el universal primer programa en cualquier lenguaje de programación, que es un programa que simplemente muestra un mensaje de saludo en la pantalla. Los navegadores web, ya se ha mencionado, disponen de un intérprete de javaScript integrado. Vamos a utilizar **Google Chrome** como intérprete para nuestros programas. **Chrome** dispone de una consola de **javaScript** en la que podemos introducir directamente código y ejecutarlo:

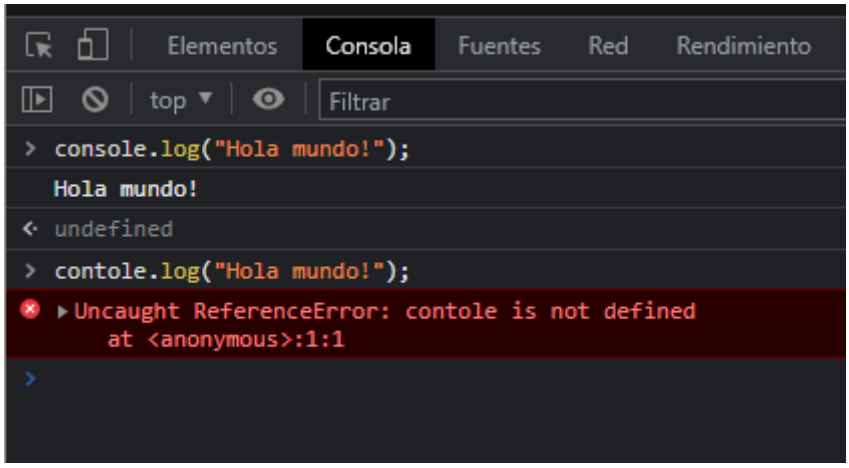

Google
google.es
Sobre Google
Tienda
Google
Nueva pestaña Ctrl + T
Nueva ventana Ctrl + N
Nueva ventana de incógnito Ctrl + Mayús + N
Historial
Descargas Ctrl + J
Marcadores
Zoom – 100 % +
Imprimir... Ctrl + P
Enviar...
Buscar... Ctrl + F
Más herramientas
Editar Cortar Copiar Pegar
Configuración
Ayuda
Salir
Guardar página como... Ctrl + S
Crear acceso directo...
Asignar nombre a ventana...
Borrar datos de navegación... Ctrl + Mayús + Supr
Extensiones
Administrador de tareas Mayús + Esc
Herramientas para desarrolladores Ctrl + Mayús + I
España
Emisión neutra de carbono desde el 2007
Publicidad
Empresa
Cómo funciona la Búsqueda
Privacidad
Términos
Configuración



Hemos accedido en **Herramientas para desarrolladores (Ctrl+May+I)** a la solapa **Consola.** Ya podemos escribir instrucciones javaScript. Cada vez que escribamos una instrucción debemos pulsar *Enter* para que se ejecute. Si la sentencia escrita tiene algún error en cuanto a sintaxis, el intérprete nos mostrará un mensaje de error indicándonos cual fue el error detectado. Vamos a ejecutar una instrucción javaScript para mostrar en la consola el mensaje *Hola mundo!*. Escribimos: **console.log("Hola mundo!);** y pulsamos *Enter*

Vemos que la instrucción se ha ejecutado correctamente y que a continuación de la instrucción se ve el mensaje. El intérprete se pone en espera de recibir una nueva instrucción.

Si la instrucción se hubiera escrito de forma incorrecta, por ejemplo: **contole.log("Hola mundo!);** veríamos:



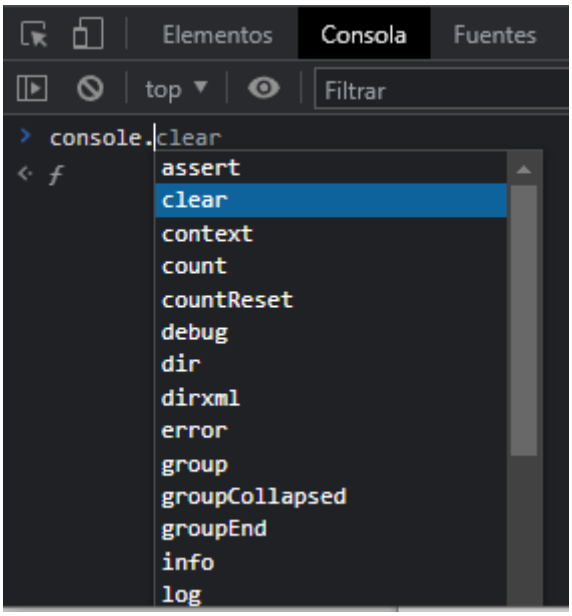
Nos dice que la instrucción **contole** no está definida, no la entiende, y por tanto no puede ejecutarla.

Esta primera instrucción de javaScript ya nos da una idea de como son. Una instrucción empieza casi siempre con un **verbo** que indica cual es el propósito de la instrucción, en este caso, mostrar en la consola: **console.log**. A continuación del verbo pueden aparecer los parámetros que particularizan la funcionalidad de la instrucción. En nuestro ejemplo, el parámetro es "**Hola mundo**" que indica que lo que se debe sacar en la consola es ese texto. Se ve que en la consola no han salido las dobles comillas. **"Hola mundo"** es lo que se llama *constante alfanumérica*, texto fijo, y ha de ir encerrado entre comillas dobles o simples. Si quisiéramos sacar en la consola el texto *Buenos días!*, deberíamos haber escrito: **console.log("Buenos días!");**

Los parámetros se escriben entre paréntesis después del verbo y separados por coma si hubiera más de uno.

Las instrucciones de javaScript deben terminar en **punto y coma** aunque no es imprescindible si el intérprete puede deducir que la instrucción terminó. Terminaremos como norma todas las instrucciones con punto y coma.

El verbo, en nuestro ejemplo: **console.log**, es en realidad **log.** La referencia **console.** Indica que el verbo **log**, en terminología informática **método**, forma parte de un conjunto de elementos de **console.** Se dice que **console** es un objeto que contiene verbos o métodos que realizan acciones sobre la consola y también tiene valores o propiedades referidas a ella. Así, si en la consola del navegador, en una línea escribimos **console.** se desplegará una lista que nos ayudará para seguir escribiendo correctamente la instrucción con los métodos y propiedades del objeto **console**. La clase **console** no tiene propiedades, pero vemos por ejemplo un método **clear** que sirve para limpiar la consola:

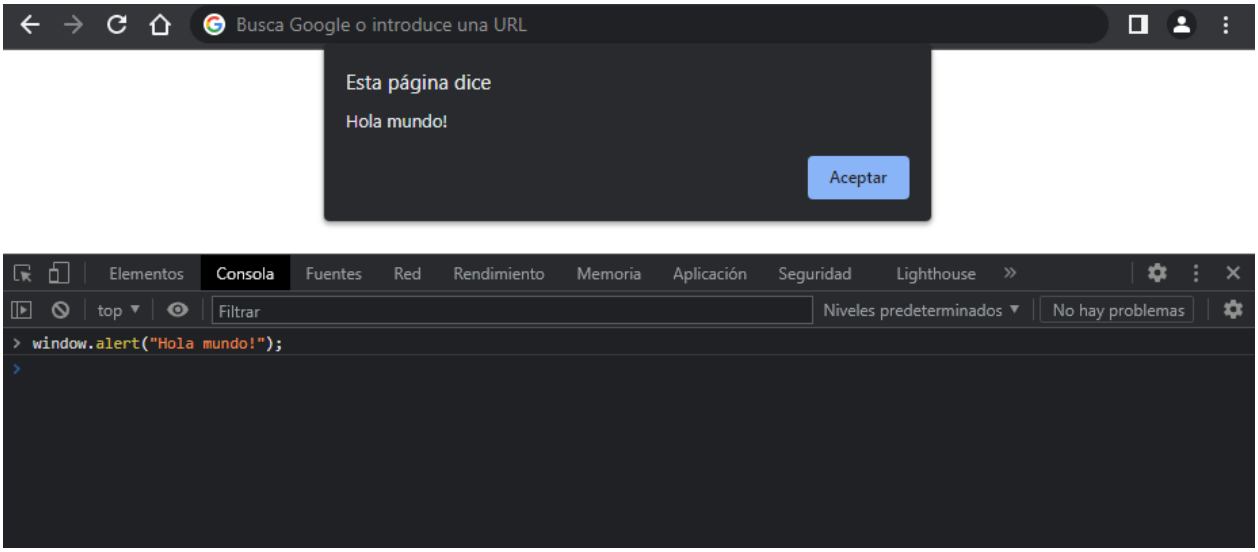


El concepto de **objeto** en informática es similar al de objeto en la vida real. Un coche es un objeto, tiene propiedades, por ejemplo, su número de matrícula, la cilindrada, el número de puertas, y también tiene métodos, un coche frena, acelera, arranca, enciende luces …

En terminología informática, se habla de **clase** para referirse a los planos del objeto, es decir, la **clase** define qué **propiedades** y **métodos** van a tener las reproducciones. Los planos del coche indican que forma y qué va a poder hacer el coche cuando fabriquemos uno. Cuando fabricamos un coche a partir de los planos se dice que tenemos una **instancia**. Una instancia tiene valores de las propiedades distintos a los de otra instancia. Si construimos dos *Ferrari Testarrosa*, tendrán las mismas propiedades, matrícula y color por ejemplo, y los mismos métodos, arrancar y frenar por ejemplo puesto que han sido construidos a partir de los mismos planos, pero tendrán distinto valor de matrícula y distinto valor de color de tapicería. Ambos serán instancias de la misma clase.

En javaScript se utiliza la terminología punto para referirse a las propiedades y métodos de una clase o de una instancia. La instrucción **console.log** hace referencia al método **log** de un objeto **console** que es la instancia creada de forma automática por el navegador de una clase a la que no tenemos acceso y por lo tanto no podremos instanciar de nuevo. Más adelante veremos como crear nuevas instancias de clases predefinidas y como crear clases nuevas y crear instancias a partir de ellas.

En programas javaScript en entorno del navegador o entorno web no se suele utilizar, en general, el método **log** de la clase **console** para mostrar información al usuario dado que si no tiene visible la consola no verá el mensaje. Si queremos mostrar el mensaje sobre una caja de alerta sobre la página web utilizaremos el método **alert** del objeto **window** instanciado también de forma automática por el navegador. El programa *Hola mundo!* quedaría:



El objeto **window** es el objeto más importante que nos ofrece el navegador para poder interactuar desde javaScript con la página cargada. Al conjunto de objetos ofrecidos por el navegador al programador para su manejo a través de javaScript, que son instanciados de forma automática, se le denomina **DOM (D**ocument **O**bject **M**odel**).** De hecho, cualquier referencia a una propiedad o método que no pueda ser resuelto de manera más inmediata será buscado dentro del objeto **window** [1]. Así si en lugar de escribir **window.alert("Hola mundo!");** hubiéramos escrito **alert("Hola mundo!");** habríamos obtenido el mismo resultado.

1 En javaScript, como norma, los nombres de instancias de objetos se escribirán en minúsculas y haciendo uso de la convención **lowerCamelCase**, y las clases de escriben también en minúsculas pero haciendo uso de la convención **UpperCamelCase** con la primera letra en mayúscula. Por ejemplo **window** hará referencia a la instancia y **Window** hará referencia a la clase. La convención **camelCase** se aplica a nombres con palabras compuestas, por ejemplo, alturamaximaactual. El nombre se debe a que las mayúsculas a lo largo de una palabra en camelCase se asemejan a las jorobas de un camello. La primera letra de cada palabra se pone en mayúscula. Existen dos tipos de camelCase:

- **UpperCamelCase** (más conocido como PascalCase), la primera letra de todas y cada una de las palabras es mayúscula. Ejemplo: **AlturaMaximaActual**.
- **lowerCamelCase** (o simplemente camelCase), igual que la anterior con la excepción de que la primera letra es minúscula. Ejemplo: **alturaMaximaActual**.

Datos

Hay varios conceptos que se deben conocer antes de poder escribir instrucciones para un programa de ordenador. Ya se ha comentado que un programa lo que hace es tomar una serie de datos, transformarlos y obtener otros datos que darán lugar posiblemente a acciones sobre otros dispositivos o sobre el mismo procesador. Vamos a ver entonces qué tipos de datos se pueden usar en javaScript, como se definen y como se usan.

Los **datos básicos** que se manejan en un ordenador, independientemente del lenguaje de programación, son siempre: datos numéricos, datos alfanuméricos y datos booleanos.

Los **datos numéricos**, como su nombre indica, son datos con los que se pueden realizar operaciones aritméticas y pueden ser escritos de diferentes formas según sean enteros o no, positivos o negativos, en base diez u otra …. Aunque se pueden escribir de diferentes formas van a ser almacenados en la memoria, en javaScript, de una única forma: según la norma **IEEE 754** o en coma flotante de doble precisión 64 bits, lo que limita el valor de los datos numéricos enteros a valores comprendidos entre $-(2^{53} - 1)$ y $+(2^{53} – 1)$ sin perder precisión, máximo 14 dígitos, y en coma flotante a $\pm(2^{-1024}$ a $2^{1024})$. Los siguientes son ejemplos de datos numéricos para su uso en javaScript:

```
23          // valor 23 entero positivo
-54         // valor 54 entero negativo
-2.4505     // valor 2.4505 en coma flotante negativo
-0.24505e+1   // el mismo en notación científica
0b10111     // valor 23 entero positivo, en binario (empieza por 0b)
027         // valor 23 entero positivo en octal (empieza por 0)
0x17        // valor 23 entero positivo en hexadecimal (empieza por 0x)
```

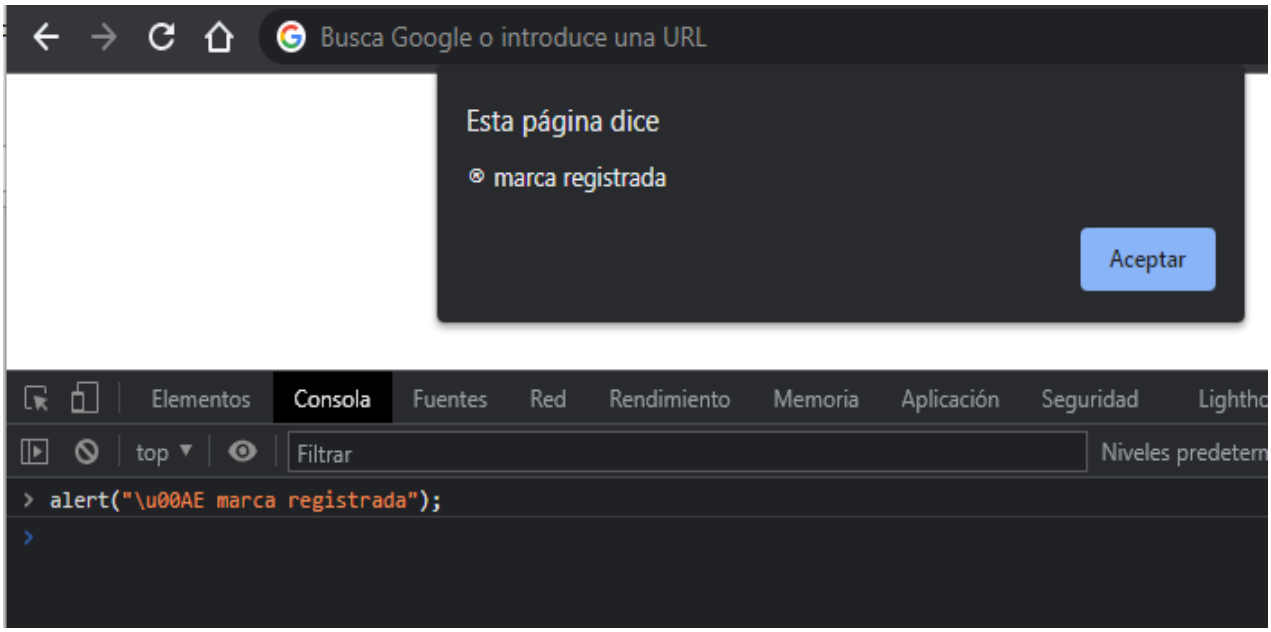
La representación de números en binario, octal o hexadecimal solo admite valores enteros.
En la actualidad hay un nuevo tipo de datos que permite rebasar el límite anterior de los valores enteros. Es el tipo **BigInt** y es de precisión arbitraria. Para indicar que un valor entero debe ser evaluado como un entero de tipo **BigInt** se añade una **n** al final del número. Por ejemplo **1024n** indica que el valor **1024** debe ser entendido como de tipo **BigInt**. **BigInt** no puede interaccionar con otros tipos de datos en la misma expresión, así la expresión **23+14** es válida en javaScript, la expresión **23n+14** no es válida, y **23n+14n** sí lo es.

Los **datos alfanuméricos** son cadenas de caracteres encerradas entre comillas dobles **"** o comillas simples **'** . El carácter comilla de apertura debe ser el mismo que el de cierre por lo que la cadena **"Había una vez'** es incorrecta, sin embargo **"Había una vez"** y '**Había una vez'** son correctas. javaScript almacena las cadenas de texto codificando los caracteres en formato **UTF-16** que no es más que una forma de almacenar el código **Unicode** que permite asignar un código numérico a cualquier carácter de cualquier idioma universal, permitiendo también asignar códigos a caracteres gráficos, por ejemplo a los emoji de whatsapp. [2]

Dentro de una cadena se pueden escribir caracteres que tienen un significado especial. Todos van precedidos de un backslash ****. Por ejemplo, dentro de una cadena se puede escribir un carácter del

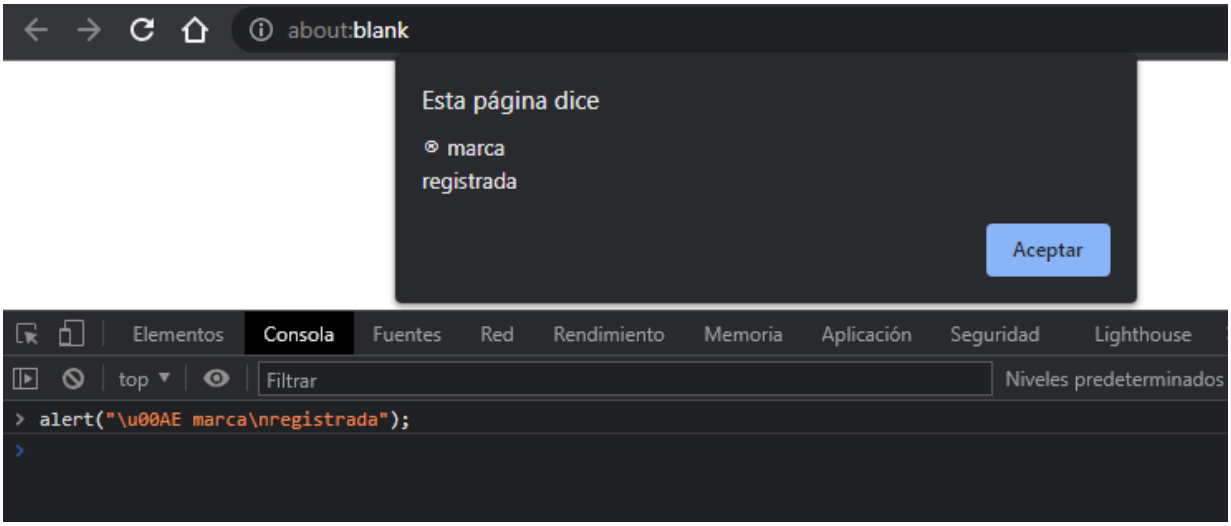
2 Tutorial sobre Unicode https://www.manjarr.es/unicode y referencia en https://www.compart.com/en/unicode/

plano básico **BMP** de Unicode por su código precediendo el número por **\u** así, por ejemplo, podremos escribir el carácter de *marca registrada*: **\u00AE**



Si en una cadena se escribe **\n** indicará que en esa posición aparecerá un retorno de carro por lo que los caracteres siguientes aparecerán en otra línea.[3]

Si en el ejemplo anterior se hubiera escrito: **alert("\u00AE marca\nregistrada");** se habría mostrado:

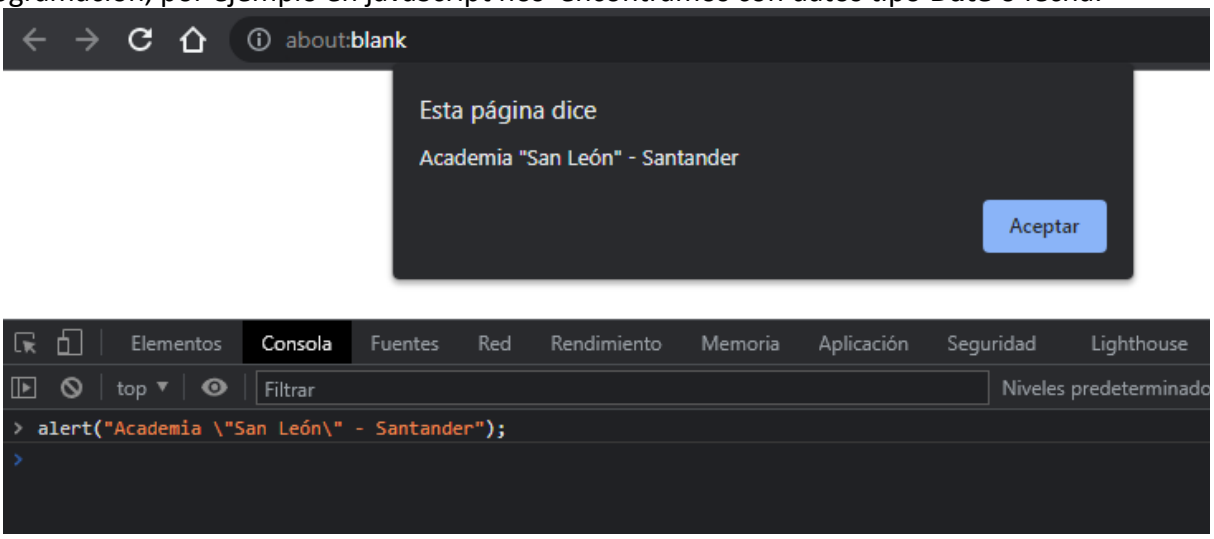


Si dentro de una cadena encerrada entre comillas dobles quisiéramos incluir unas comillas dobles, debemos preceder estas comillas con un backslash. En terminología informática se habla de *escapar* las comillas: "**Academia \"San León\" - Santander".**

3 No confundir con el salto de línea en las páginas web que es la marca **
. Si en la página se muestra **\n no tendrá ningún efecto de salto de línea.

Los **datos lógicos o booleanos** solo son dos: **true** o **false**. Veremos más adelante como se realizan operaciones con datos aritméticos y alfanuméricos. Veremos que también se pueden realizar operaciones lógicas como por ejemplo si un valor numérico es mayor que otro. Estas expresiones son ciertas o falsas (**true** o **false**).

Nos podemos encontrar con tipos de datos más complejos que varían en función del lenguaje de programación, por ejemplo en javaScript nos encontramos con datos tipo **Date** o fecha.



Una cadena en javaScript puede tener hasta 2^{53} - 1 caracteres.[4]

Expresiones[5]

Una **expresión** es una **unidad de código** que al ser evaluada devuelve un **valor**. Por ejemplo: **(5 * 2)/(4-2)** es una expresión de tipo aritmético que al ser evaluada devuelve un valor, en el ejemplo **5**. En función de los tipos de datos y operadores que participen nos podemos encontrar con expresiones **aritméticas**, **relaciones** y **lógicas** , combinándose muchas veces entre sí formando expresiones complejas. El valor devuelto por una expresión será de uno de los tipos mencionados. Un expresión estará formada por los datos y los operadores que indican que tipo de operación hay que realizar con ellos.

Operadores aritméticos

Son los operadores que participan en las **expresiones aritméticas**. Una expresión aritmética al ser evaluada siempre devuelve un **número**.

4 Referencia String en MDN Mozilla
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String

5 Referencia de operadores y expresiones en MDN Mozilla
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators##aritm%C3%A9ticos

Operador	Descripción	Ejemplo
++ Incremento ()	Operador unario. Agrega uno a su operando, que debe ser el nombre de una variable. Si se usa como operador prefijo (**++x**), devuelve el valor de su operando después de agregar uno; si se usa como operador sufijo (**x++**), devuelve el valor de su operando antes de agregar uno.	Si x es 3, ++x establece x en 4 y devuelve 4, mientras que x++ devuelve 3 y , solo entonces, establece x en 4.
-- Decremento	Operador unario. Resta uno de su operando. El valor de retorno es análogo al del operador de incremento.	Si x es 3, entonces `--x` establece x en 2 y devuelve 2, mientras que `x--` devuelve 3 y, solo entonces, establece x en 2.
- Negación unaria	Operador unario. Devuelve el valor negativo de su operando.	Si x es 3, entonces `-x` devuelve -3.
+ Positivo unario	Operador unario. Intenta convertir el operando en un número, si aún no lo es. Devuelve el valor positivo del número	`+"3"` devuelve `3`. `+true` devuelve `1`.
+ Suma	Operador binario. Devuelva la suma de los dos operandos	5 + 8 devuelve 13
- Resta	Operador binario. Devuelve la resta de los dos operandos	8 - 4 devuelve 4
* Multiplicación	Operador binario. Devuelve la multiplicación de sus dos operandos	8 * 4 devuelve 32
/ División	Operador binario. Devuelve el resultado de dividir el primer operando dividendo entre el segundo cociente	12 / 5 devuelve 2.4
% Resto	Operador binario. Devuelve el resto entero de dividir los dos operandos.	12 % 5 devuelve 2.
** Exponenciación	Calcula $base^{exponente}$	`2 ** 3` returns `8`. `10 ** -1` returns `0.1`.

Se pueden utilizar los operadores **( )** para modificar la precedencia en la evaluación de la expresión.

La siguiente tabla muestra la precedencia en los operadores aritméticos, en orden descendente, cuando aparecen en la misma expresión:

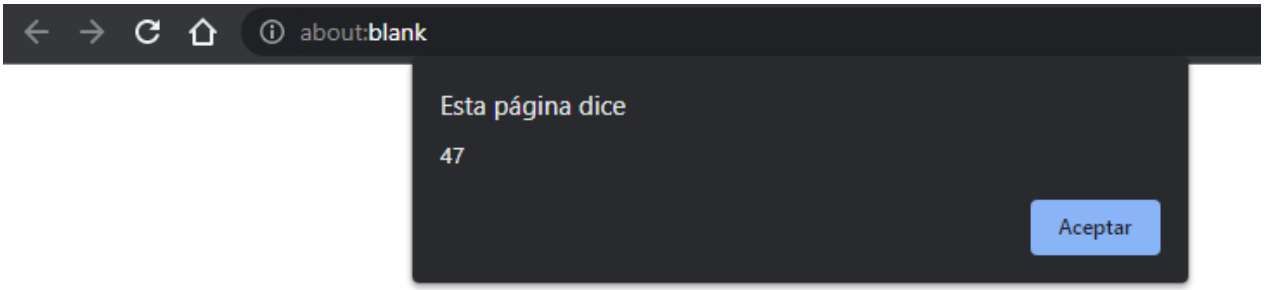
Operadores
( )
- + ++ --
*** / %**
+ -

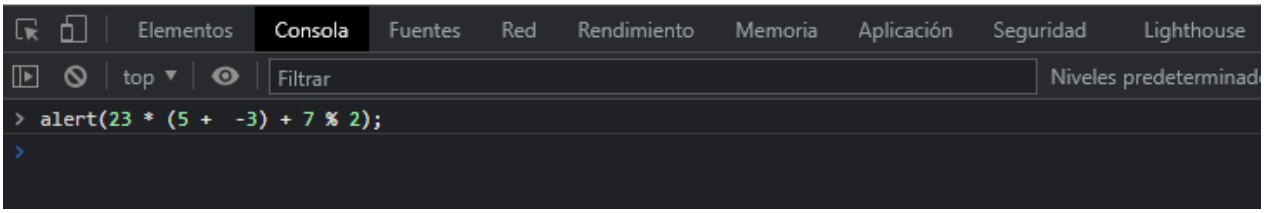
Por ejemplo, el orden de evaluación de la siguiente expresión aritmética compuesta:

```
23*(5+ -3)+7%2
```

será:

1. Primero se evaluará la expresión entre paréntesis **(5 + -3)** al ser los paréntesis los que más precedencia tienen.
2. Dentro de esta expresión se aplicará el operador unario **–** al **3**
3. Se evaluará la expresión **5 + -3** dando como resultado **2.**
4. Tenemos ahora la expresión: **23 * 2 + 7 % 2**. Los operadores ***** y **%** tienen la misma precedencia evaluándose de izquierda a derecha, por lo que primero se evaluará la expresión **23 * 2**, dando **46**
5. Nos queda ahora la expresión **46 + 7 % 2**. Se evaluará ahora la expresión **7 % 2** por tener más precedencia el operador **%** que el **+**. Dará como resultado **1**.
6. Queda la expresión **46 + 1** que da como resultado **47.**





Operadores relacionales

Los operadores relacionales permiten comparar dos operandos y devuelven un valor **booleano**: **true** o **false**.

Operador	Descripción
<	Operador menor que. Indica si el primer operando es menor que el segundo
>	Operador mayor que. Indica si el primer operando es menor que el segundo
<=	Operador menor o igual que. Indica si el primer operando es menor o igual que el segundo
>=	Operador mayor o igual que. Indica si el primer operando es mayor o igual que el segundo
==	Operador igual que. Indica si el primer operando es igual que el segundo

===	Operador estrictamente igual que. Indica si el primer operando es igual en tipo y valor que el segundo
!=	Operador distinto que. Indica si el primer operando no es igual que el segundo
!==	Operador estrictamente igual que. Indica si el primer operando es no igual que el segundo o en tipo o en valor o ambos

Todos los operadores tienen la misma precedencia.

Por ejemplo, la expresión **5 > 8** devolverá el valor booleano **false**. Si en una expresión aparecen expresiones aritméticas y relacionales, se evaluarán antes las aritméticas, así en la expresión **5 > 10 – 6,** primero se evaluará **10 - 6** que tendrá como resultado **4** y luego se comparará con **5** dando como resultado el valor booleano **true**.

Operadores booleanos o lógicos

Se corresponden con las operaciones booleanas **NOT, AND** y **OR** y sus tablas de verdad[6]
En javaScript estos operadores son, respectivamente, **! && y ||.** El orden de precedencia es **!,** el que más, seguido de **&&** y **|| .**

Si en una expresión compleja aparecen expresiones, aritméticas y lógicas, primero se evaluarán las aritméticas, luego las relaciones y por último las booleanas.

Ejemplo:

```
((3+5)/2)>(4−4) && true || (!((3+1)>1) || false)
```

Ya sabemos que primero se evalúan las expresiones aritméticas empezando por las contenidas en paréntesis más internos, por prioridad de operador y a igualdad de izquierda a derecha, por lo tanto primero se evaluará la expresión **3 + 1**

```
((3+5)/2)>(4−4) && true || (!(4>1) || false)
```

A continuación **3 + 5**

```
(8/2)>(4−4) && true || (!(4>1) || false)
```

Ahora en igualdad anidación de paréntesis, se evaluará **8 / 2**

```
4>(4−4) && true || (!(4>1) || false)
```

Después **4 - 4**

```
4>0 && true || (!(4>1) || false)
```

Ya no quedan expresiones aritméticas, es el turno de las expresiones relacionales, primero las incluidas en los paréntesis más internos que es **4 > 1**

```
4>0 && true || (!true || false)
```

Después **4 > 0**

```
true && true || (!true || false)
```

6 Algebra de Boole https://bookdown.org/alberto_brunete/intro_automatica/algebraboole.html

Ya solo quedan expresiones booleanas. Primero las incluidas en los paréntesis más internos y por precedencia le corresponde a ! **true**

true && true || (false || false)

Después a **false || false**

true && true || false

Por precedencia le toca a **true && true**

true || false

Y finalmente el resultado de la expresión es **true**

javaScript dispone de otros operadores que se irán viendo en el documento, pero que son más específicos de este lenguaje. Los anteriores son más o menos estándar de la mayoría de los lenguajes de programación.

Variables

El concepto de **variable** es básico en cualquier lenguaje de programación. Una variable no es más que un **nombre** que representa una **zona de memoria** en la que almacenar un dato. Cuando en una expresión aparece un nombre que no se corresponde con un dato de los anteriormente vistos, lo que se hace es tomar el contenido de esa posición de memoria y sustituirlo por el nombre. Por ejemplo si hay una variable con nombre **edad** que contiene un **25**, entonces la expresión **edad * 2** será evaluada a **50.**

El nombre de una variable en javaScript debe comenzar con una **letra**, un guión bajo **_** o un signo de dólar **$**. Los siguientes caracteres también pueden ser dígitos **0** al **9**.

javaScript distingue entre mayúsculas y minúsculas, por lo tanto la variable con nombre **Edad** será distinta de **edad**.

Una variable en javaScript debe ser **declarada** antes de ser utilizada. Se declara una variable con el operador **var** o con **let.** Cuando más adelante se mencionen los **bloques de código** aclararemos cual es la diferencia entre declarar una variable con **var** o con **let.** Si se va a declarar una variable cuyo valor sabemos que no va a cambiar nunca, la declararemos como **constante** con el operador **const**.

```
var  edad;
let fechaNacimiento, mes;
const PI= 3.1416;
```

Como convenio se debe utilizar la terminología **lowerCamelCase**[9] para poner nombre a las variables. Las constantes se suelen poner en **mayúsculas**.

La asignación de un valor a una variable se hace principalmente con el operador de asignación **=**. A la izquierda se pone el nombre de la variable y a la derecha una expresión que al ser evaluada devolverá el valor que será asignado a la variable:

```
edad = 22;
mes = mes + 1
```

En javaScript hay operadores abreviados de asignación para efectuar operaciones aritméticas habituales como por ejemplo incrementar el contenido de la variable si es numérica, y también para las operaciones lógicas:

Operador	Descripción	Ejemplo
+=	Incrementa el valor de la variable en la cantidad de la expresión a la derecha	**x += 2*3**, equivale a **x = x + 2*3**
-=	Incrementa el valor de la variable en la cantidad de la expresión a la derecha	**x -= 2**, equivale a **x = x - 2**
*=	Multiplica la variable por la cantidad de la expresión de la derecha	**x *= 5** equivale a **x = x * 5**
/=	Divide la variable por la cantidad de la expresión de la derecha	**x /= 8** equivale a **x = x / 8**
%=	Asigna a la variable de la izquierda el resultado de calcular el resto de dividir la variable por el valor de la expresión de la derecha	**x %= 5** equivale a **x = x % 5**
&&=	Asigna a la variable de la izquierda el resultado de hacer la operación lógica AND a la variable de la izquierda y el valor de la expresión de la derecha	**x &&= (5 > 2)** equivale a **x = x && (5 > 2)**
||=	Asigna a la variable de la izquierda el resultado de hacer la operación lógica OR a la variable de la izquierda y el valor de la expresión de la derecha	**x ||= (5 > 2)** equivale a **x = x || (5 > 2)**

En muchos lenguajes de programación se ha de especificar, en el momento de la declaración, el **tipo** de contenido que va a tener la variable. Así en Java si tuviéramos que declara una variable para contener enteros deberíamos escribir:

```
int edad;
```

En javaScript no es así, en una variable se puede guardar cualquier tipo de contenido y guardar, por ejemplo, una cadena y a continuación un dato numérico, no da ningún problema. Esta claro que solo el último guardado permanecerá.

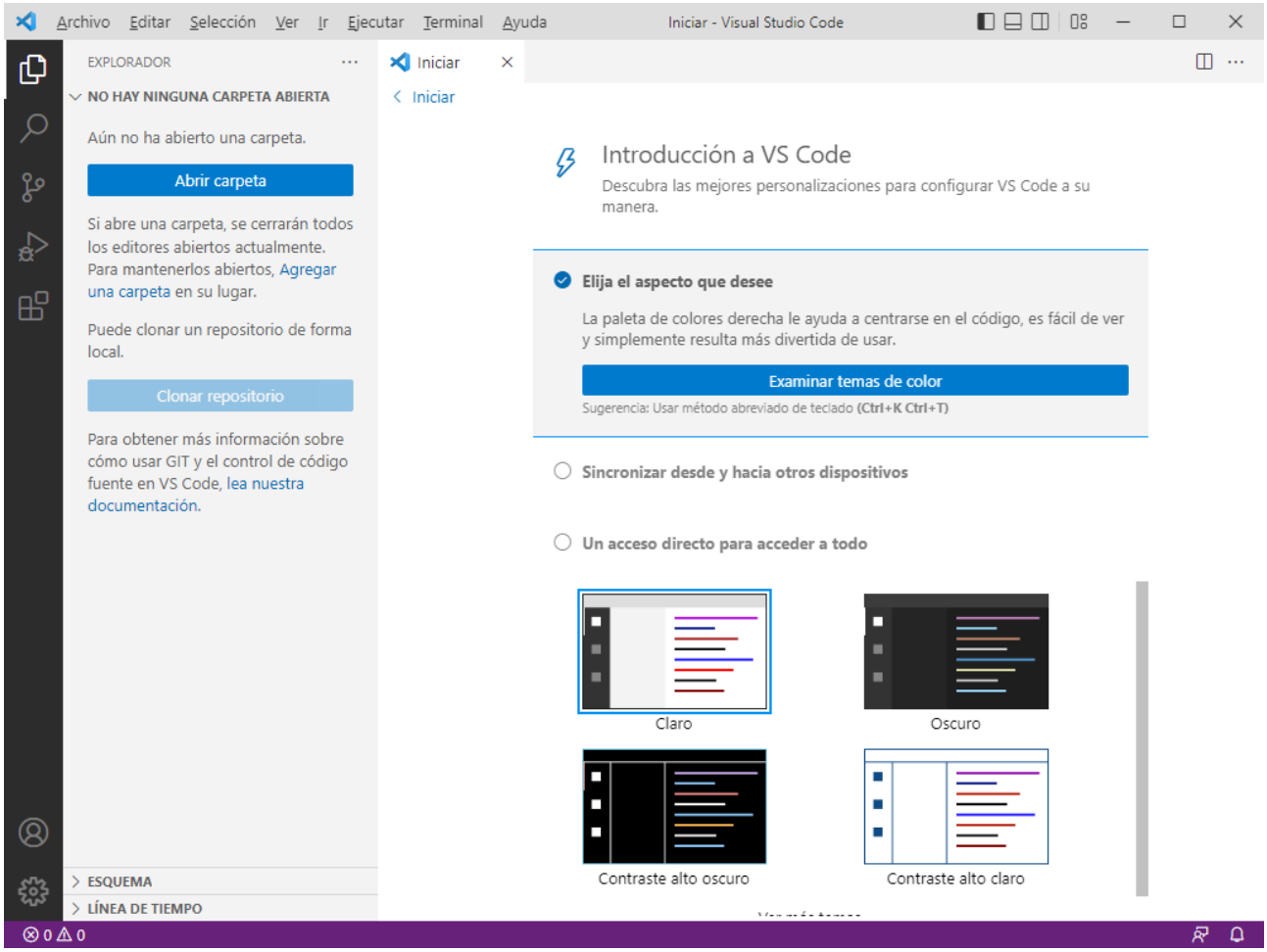
Se dice que Java es un lenguaje de **tipado estricto** y javaScript es de **tipado débil**.

Estructuras de programación.

Ya tenemos las nociones básicas para empezar a hacer programas. Normalmente un programa va a constar de multitud de instrucciones que se van a ejecutar de forma **secuencial** una tras otra.

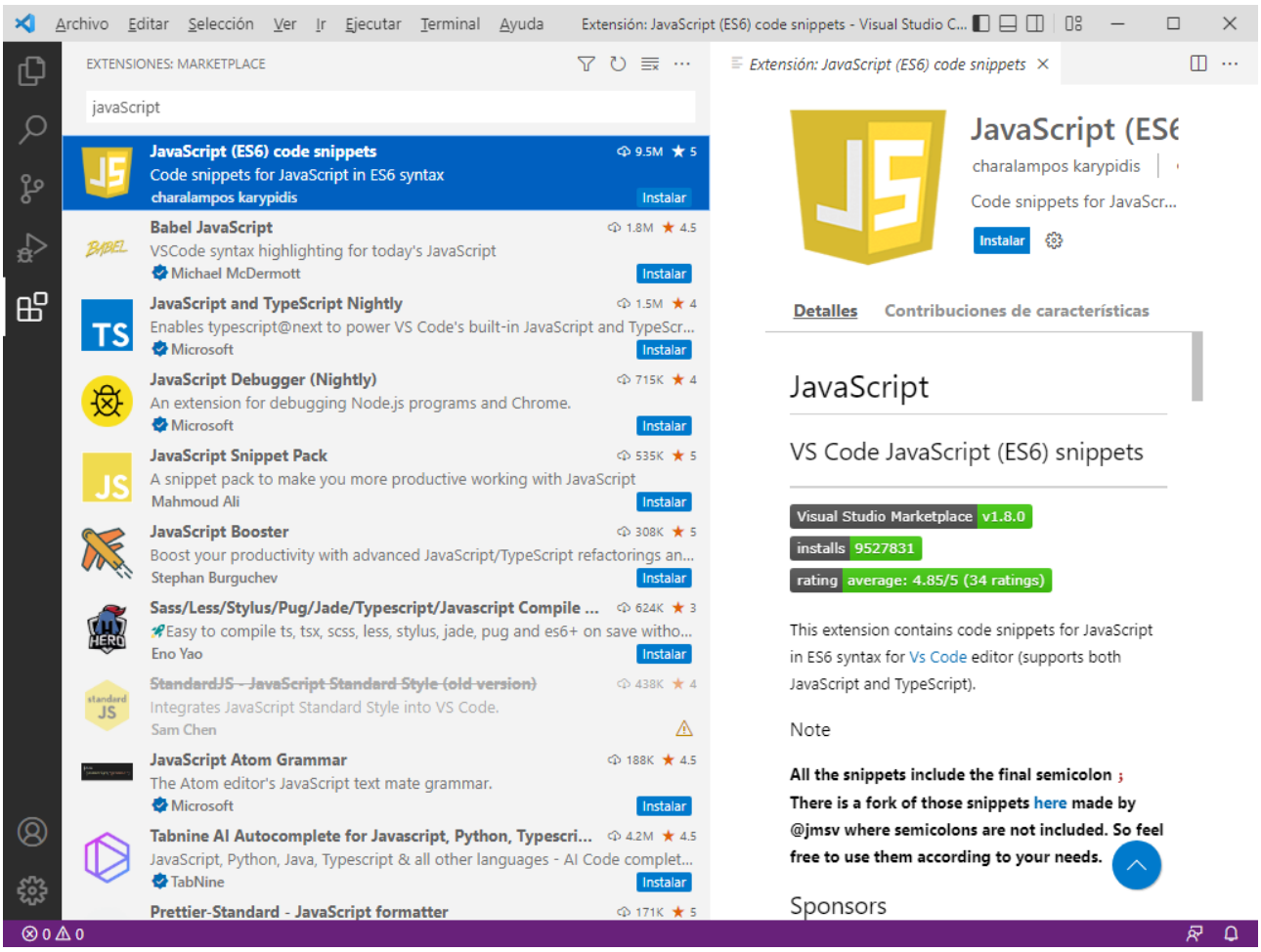
Habrá veces que querremos ejecutar unas instrucciones u otras en función de alguna casuística, por ejemplo, si la edad de la persona en menor de 18 años no querremos ofrecerle productos no aptos para menores de edad. Estas instrucciones que en función de una condición permiten la ejecución de una serie de instrucciones u otras se denominan instrucciones o **sentencias condicionales** y todos los lenguajes de programación disponen de ellas. Las iremos viendo a medida que las necesitemos.

Como un programa va a constar de varias instrucciones vamos a necesitar escribirlas en un entorno agradable que nos permita la ejecución y depuración de la ejecución. Ya lo hemos mencionado, vamos a utilizar **Visual Studio Code.** Si ya lo hemos descargado e instalado, la primera vez que lo ejecutemos nos pedirá que instalemos el paquete de idioma y reiniciemos. Pulsando el botón correspondiente todo se hará de forma automática. También nos pedirá elegir un tema de colores para la aplicación en la ventana del inicio:

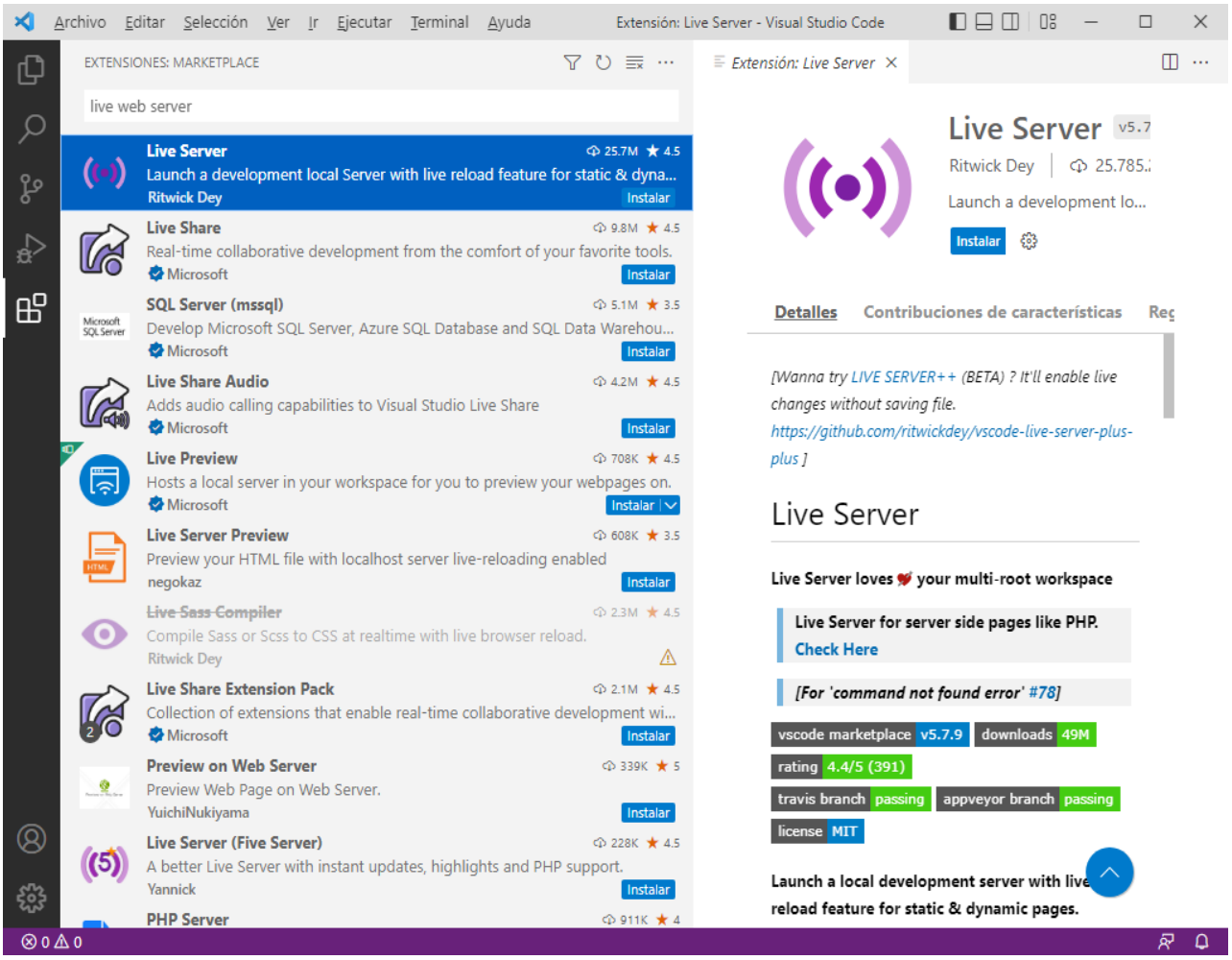


En **Visual Studio Code** las **aplicaciones** o programas se asocian a **carpetas** en el sistema de archivos. El proceso normal de creación de una aplicación es por tanto crear la carpeta y abrirla.

Antes de crear nuestra primera aplicación podemos decir a **Visual Studio Code** que instale las **extensiones** de javaScript porque **Visual Studio Code** permite trabajar con multitud de lenguajes de programación. Cerramos la solapa **Iniciar** y pulsamos en el icono  de la barra de herramientas lateral y escribimos **javascript** en la caja de texto:



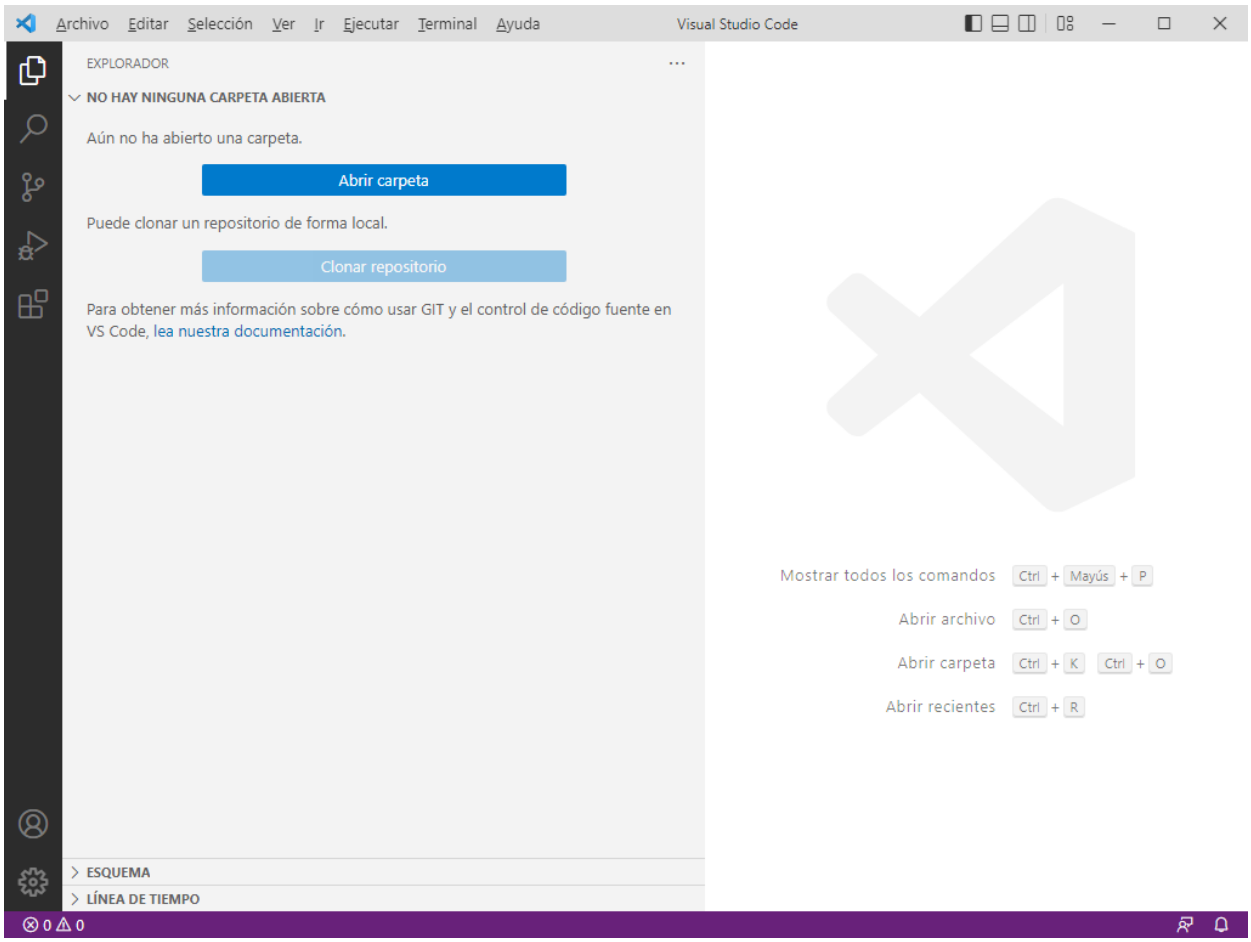
Seleccionamos e instalamos **JavaScript (ES6) code snippets**. A continuación escribimos **live web server** en la caja de texto, y seleccionamos e instalamos **Live Server:**



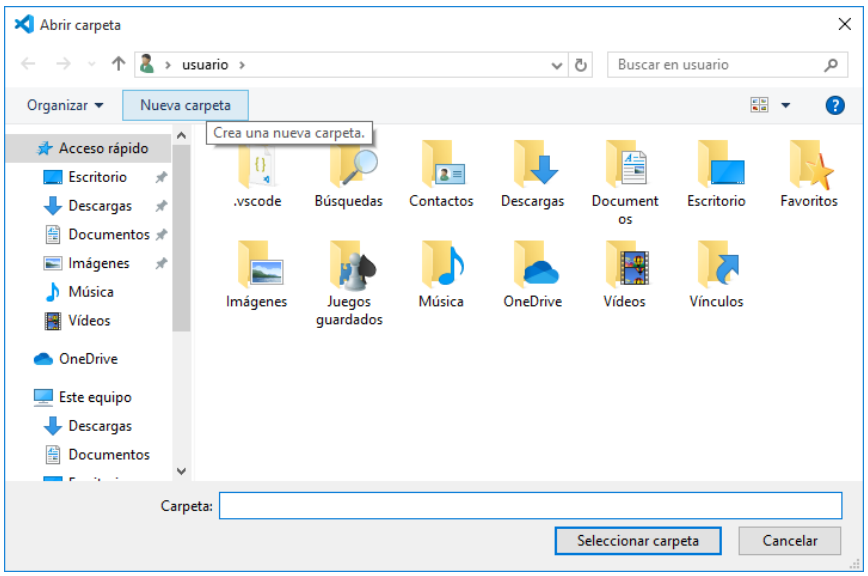
Estas extensiones **no son imprescindibles** para desarrollar en javaScript pero si nos facilitarán escribir el código y su depuración. Un **code snippet** es un trozo de código que es generado al introducir en el texto una combinación de caracteres dada seguido de **enter**. Por ejemplo si en un archivo de javaScript escribimos **clo** seguido de **enter** se insertará el código **console.log('object :>> ', object);**

Como vamos a ejecutar javaScript desde el intérprete de un navegador necesitamos incrustar el código javaScript en un página web que va a ser servida desde un servidor web. Ese es el propósito de haber instalado la extensión **Live Server.**

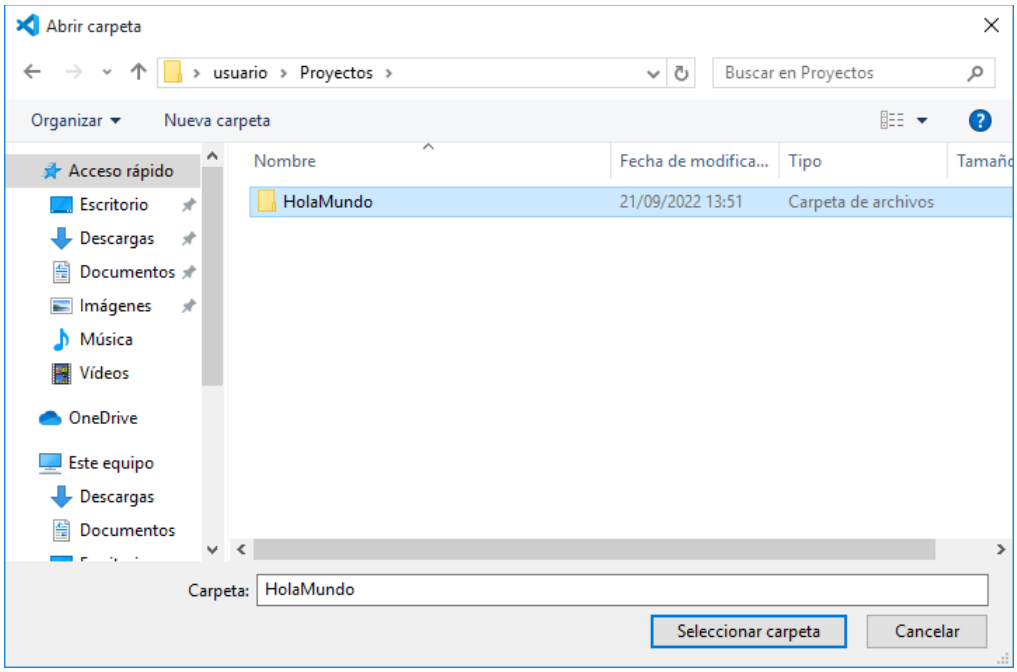
Empezamos. Vamos a crear la aplicación **Hola mundo** pero desde una página web. Tendremos que crear una carpeta y abrirla, para ello pulsamos sobre el icono  de la barra de herramientas lateral

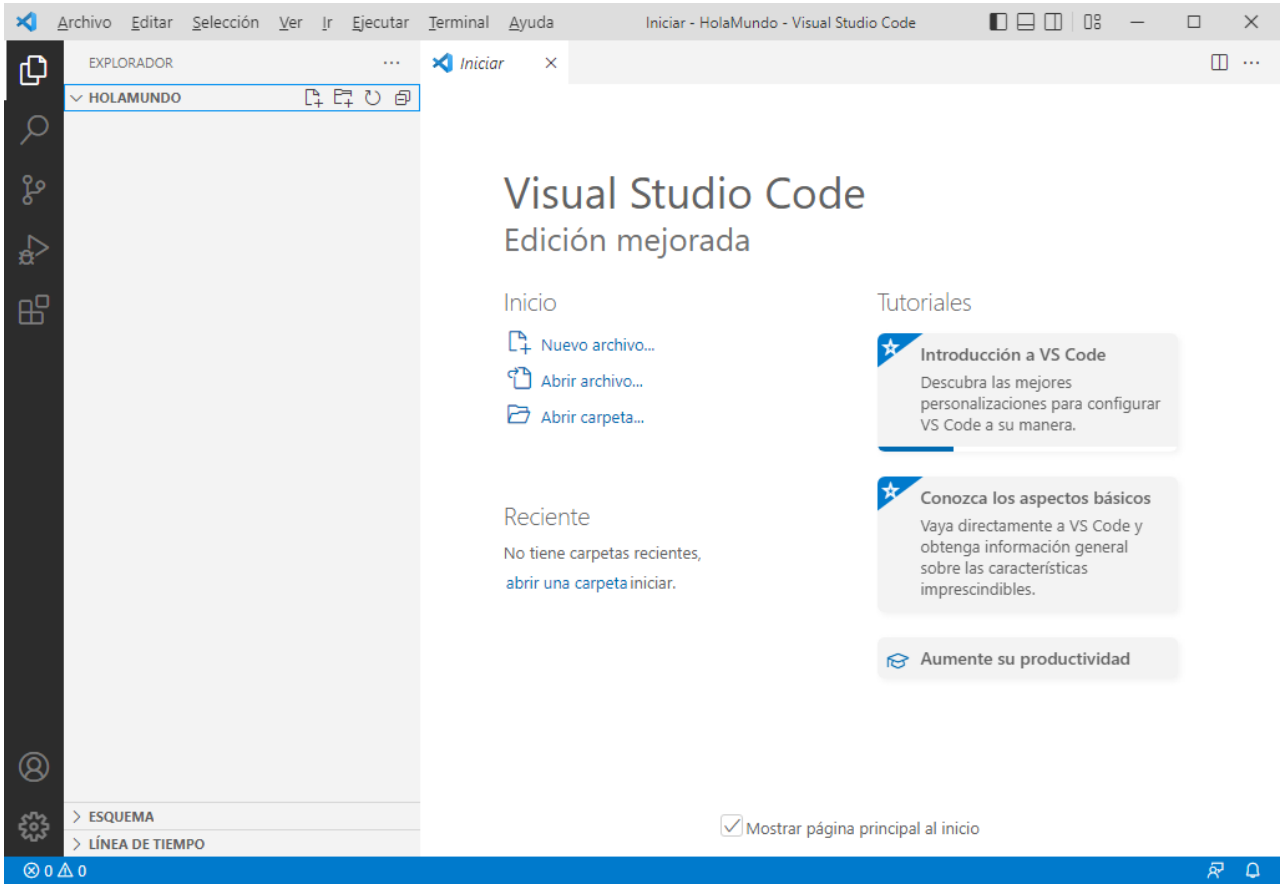


Pulsamos sobre **Abrir carpeta**

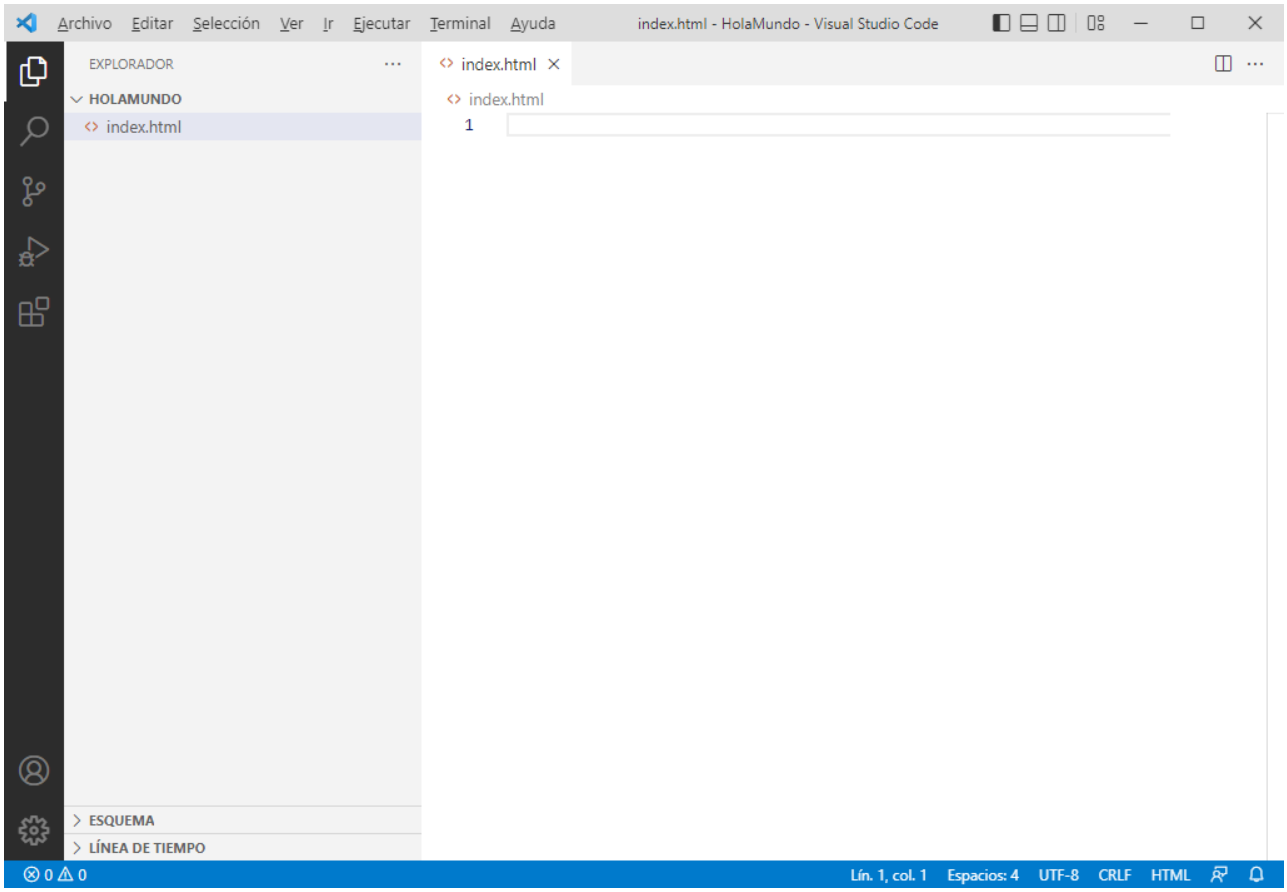


y sobre **Nueva carpeta**. Vamos a colocar todos los proyectos de **Visual Studio Code** bajo una carpeta común que se llame por ejemplo **Proyectos** y bajo esta una carpeta para este primer proyecto que vamos a llamar por ejemplo **HolaMundo** y pulsamos en **Seleccionar carpeta**.

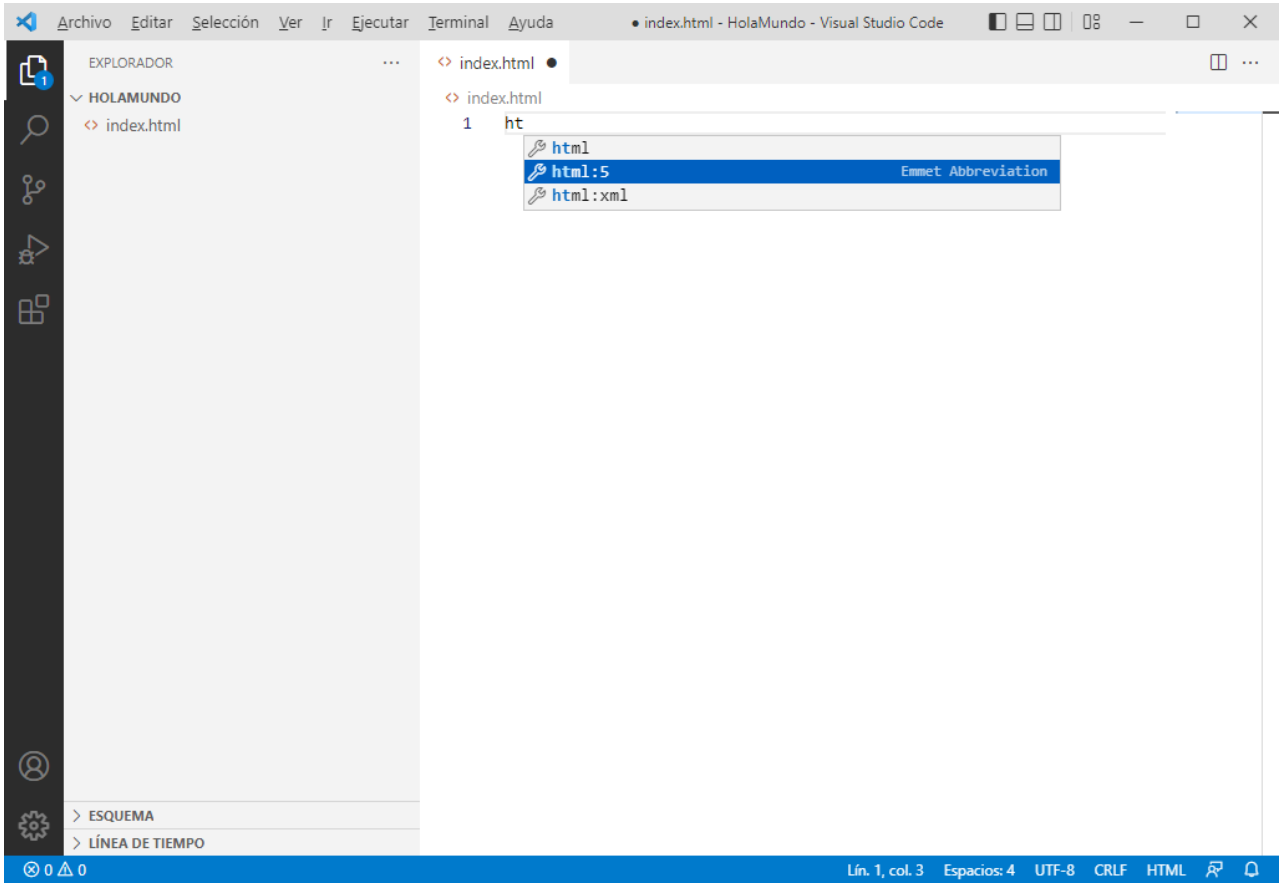




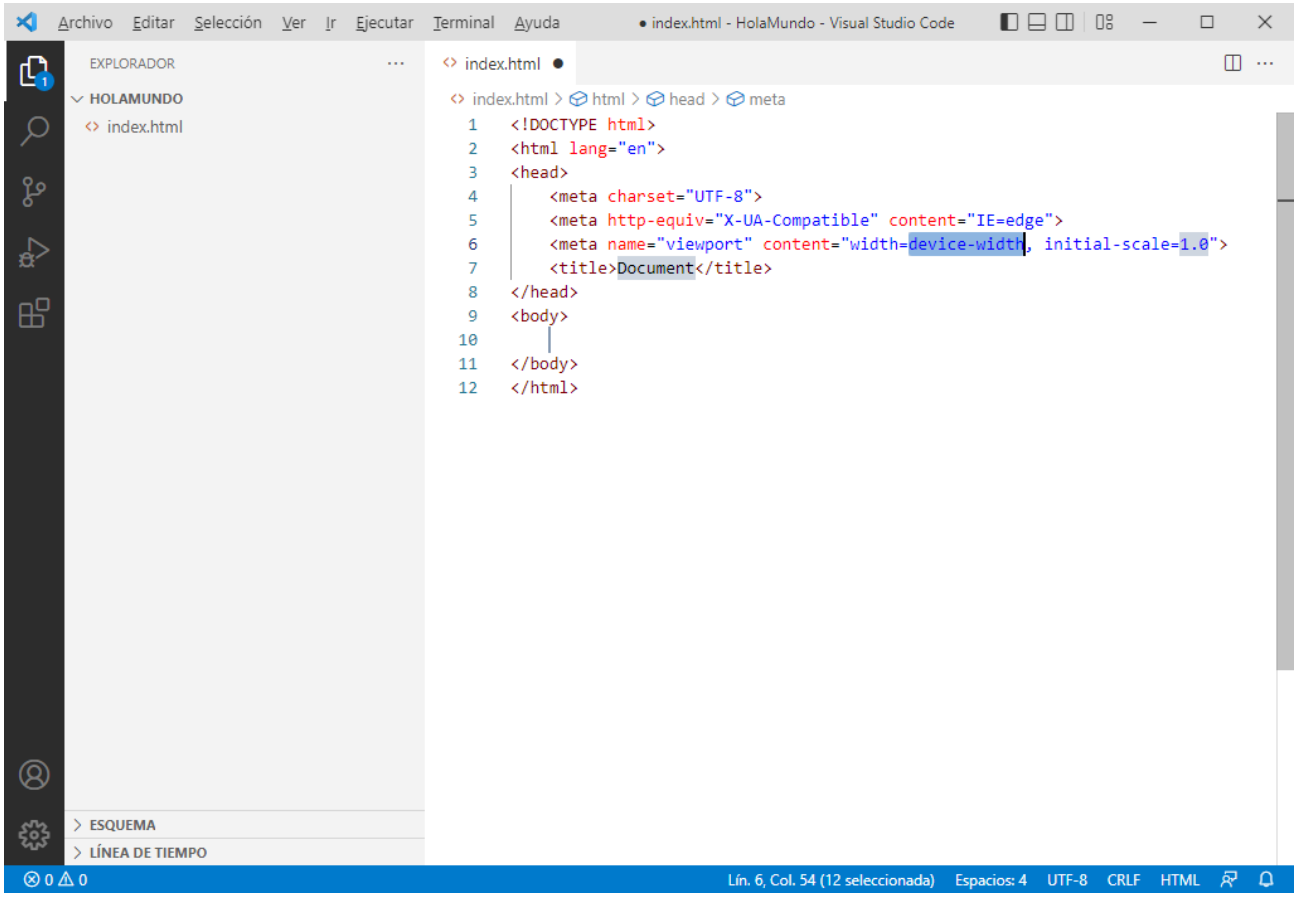
En el panel **EXPLORADOR** vamos a ver en todo momento la estructura de archivos y carpetas de nuestro proyecto, que en este momento está vacío. Vamos a crear el archivo web con extensión **.html** que nos va a permitir lanzar la ejecución del código javaScript. Existen varias opciones, pulsar en **Nuevo archivo** en la solapa de **Iniciar**, en el menú de **Archivo → Nuevo archivo**, pulsar en el icono  al lado de **EXPLORADOR,** seleccionar **Nuevo archivo** en el menú contextual que aparece al pulsar botón derecho del ratón sobre una zona vacía del panel **EXPLORADOR** … Llamaremos al archivo **index.html**



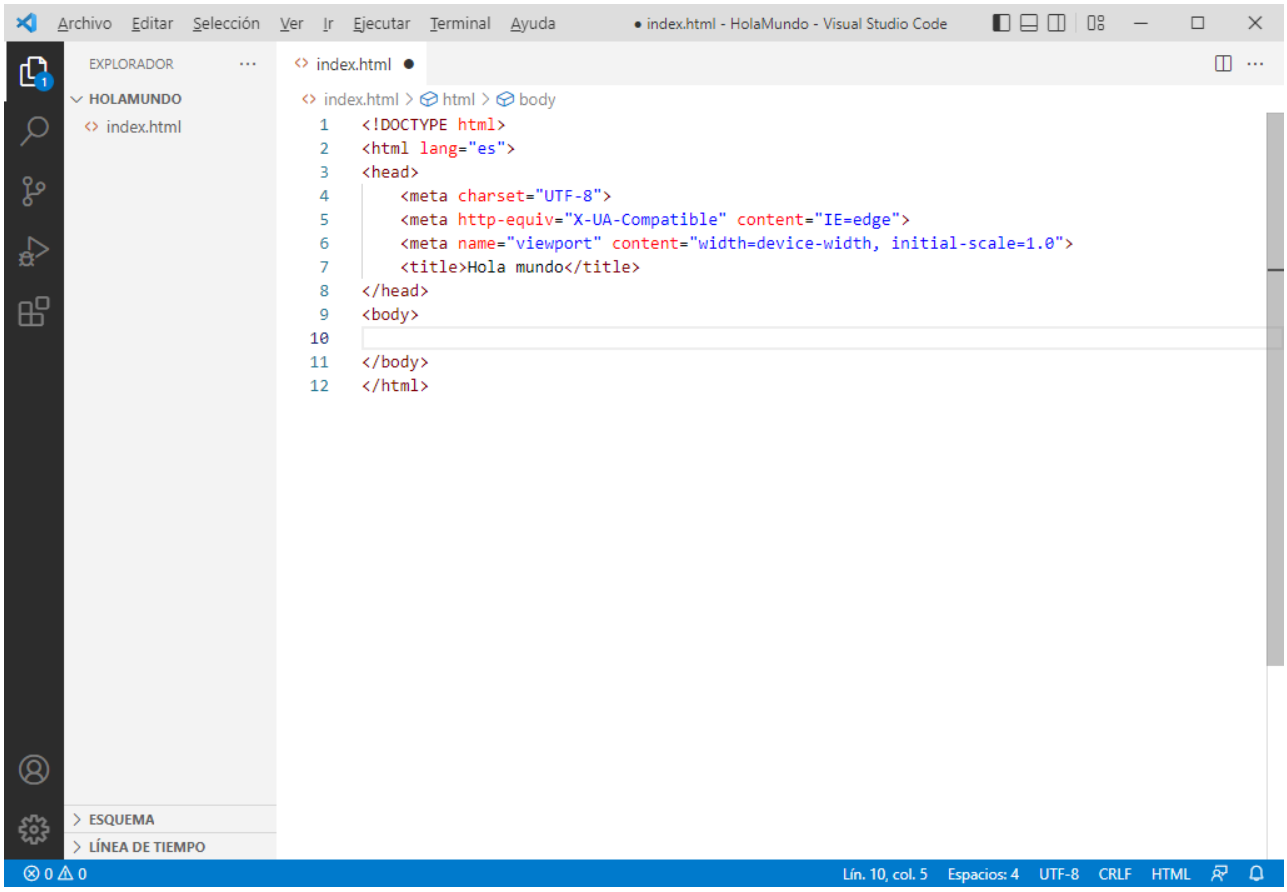
Hemos cerrado la solapa **Iniciar** porque no se va a necesitar. Si empezamos a escribir el texto **ht** dentro del archivo **index.html** veremos que se despliega una lista con opciones que el editor entiende que tienen que ver con el contexto en el que se está.



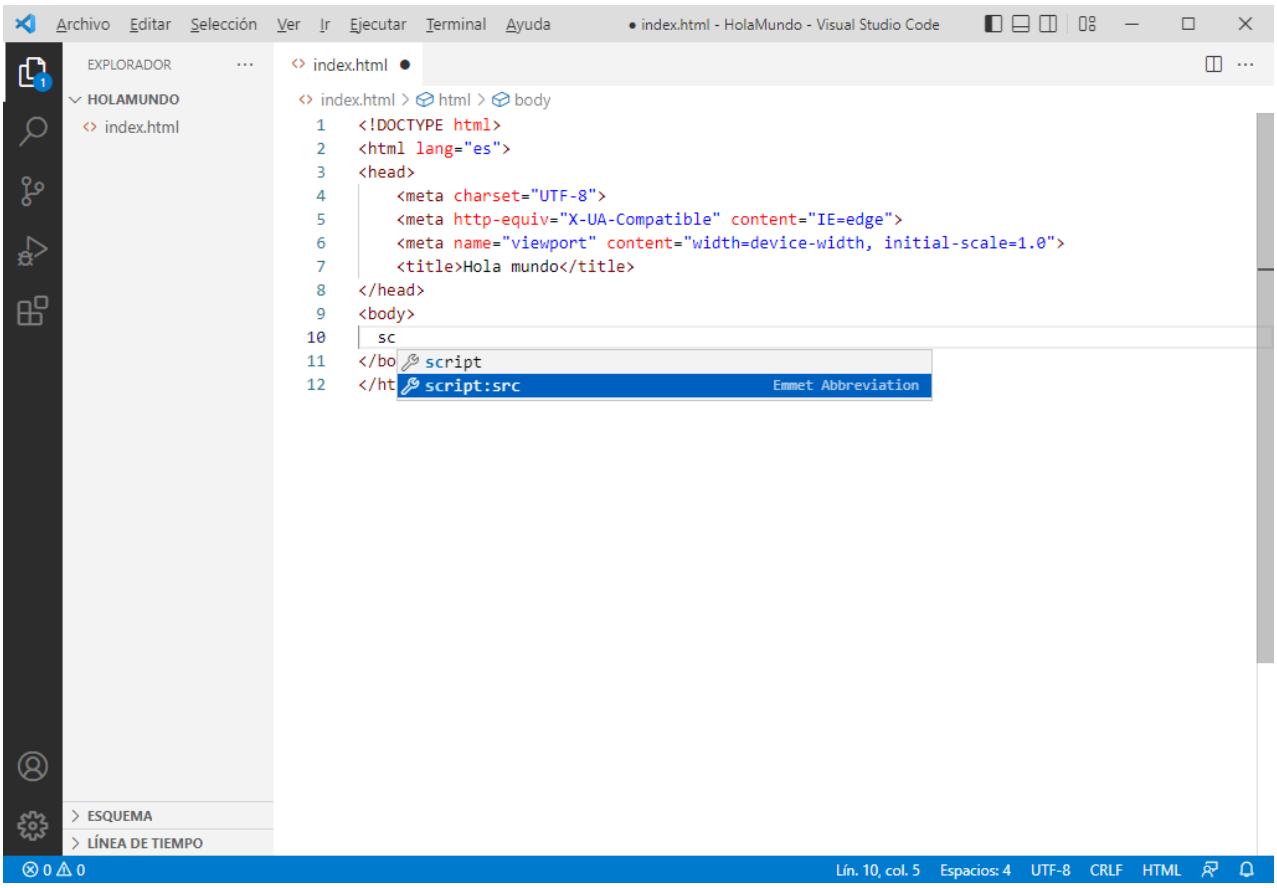
Si seleccionamos **html:5** y damos **enter** veremos que se inserta el siguiente código. Lo que hemos lanzado es un **snippet code** para html



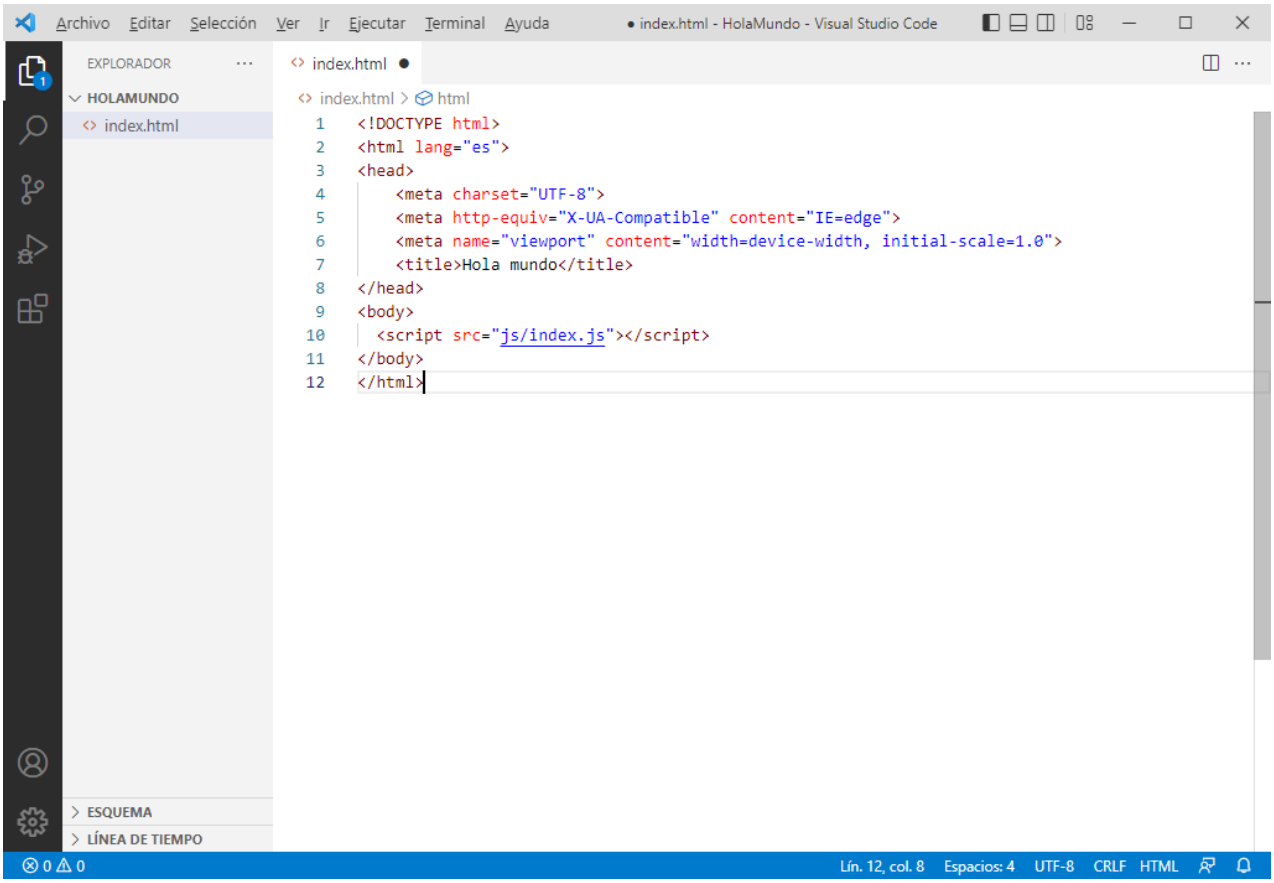
Se ha generado el código que representa el esqueleto de una página web en html 5. Cambiamos **lang="en"** por **lang="es"** y en **title** cambiamos **Document** por **Hola mundo**.



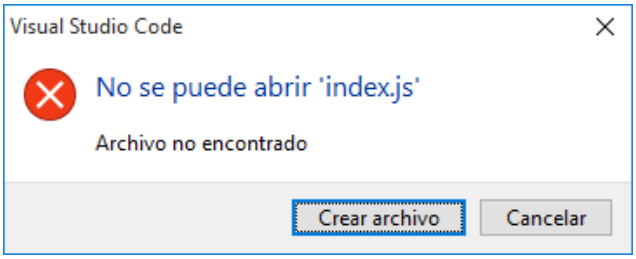
En una nueva línea en blanco justo antes de **</body>** empezamos a escribir **sc** y se desplegará una lista de ayuda al contexto en la que seleccionaremos **script:src** seguido de **enter**



Se insertará el snippet: **<script src=""></script>** Entre las comillas escribiremos **js/index.js**



Veremos que aparece subrayado el nombre del archivo. El archivo todavía no existe, así como tampoco existe la carpeta **js**. Si pulsamos la tecla **Ctrl** y hacemos clic sobre el nombre del archivo nos mostrará la caja de alerta:



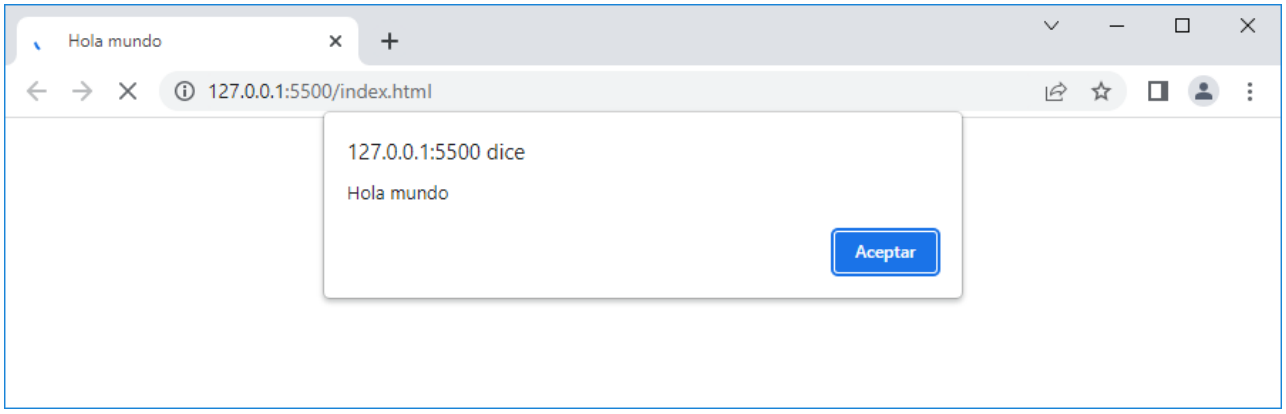
Que nos va a permitir crear ambos de forma automática.

Vemos que aparece una nueva solapa con un archivo vacío **index.js** y en la ventana **EXPLORADOR** aparece tanto la nueva carpeta **js** como el nuevo archivo **index.js**.

Ya podemos escribir nuestro código javaScript. Para guardar todos los cambios en todos los archivos pulsamos **Ctrl+K S** o seleccionamos **Archivo → Guardar todo** en la línea de menú.

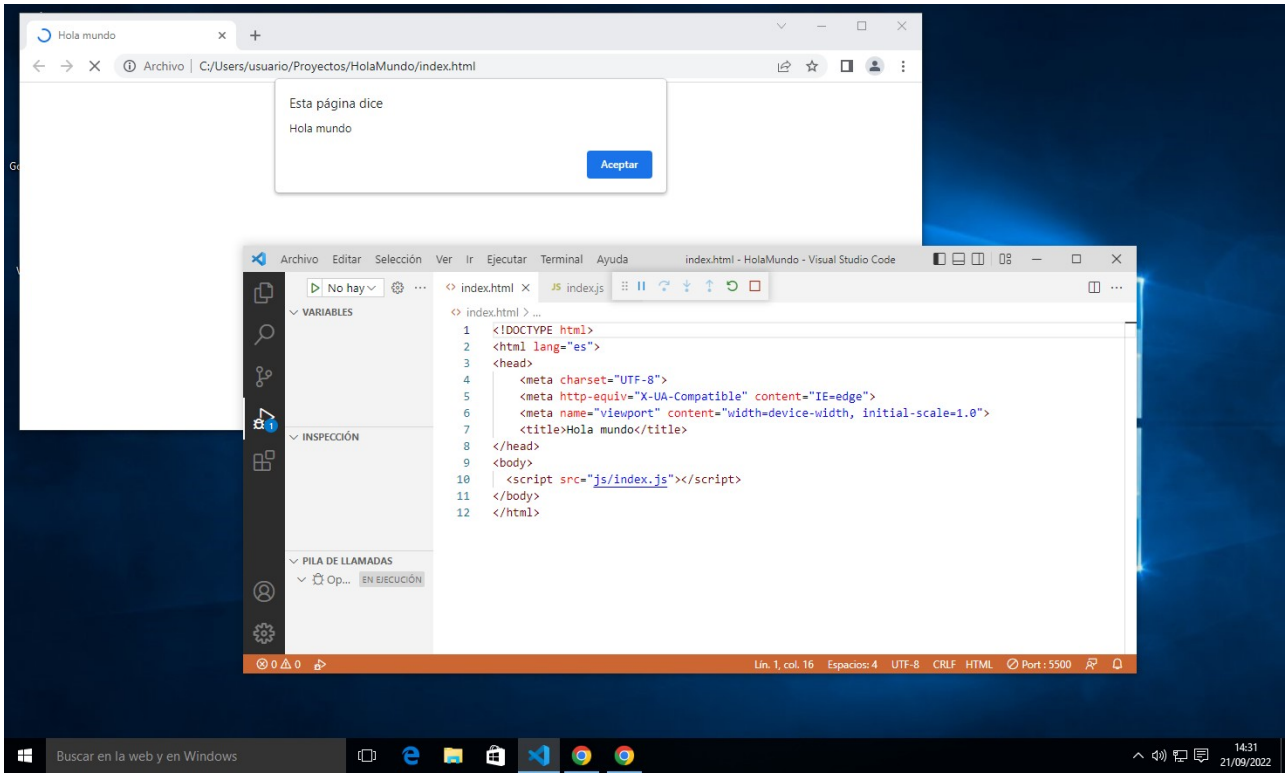
Para ejecutar el código, estando seleccionada la solapa del archivo **index.html**, iniciamos el servidor web pulsando en 

en la barra de estado (esto solo habrá que hacerlo la primera vez) . De forma automática se lanzará el navegador predeterminado mostrando la página **html** y el código **javaScript** asociado.



Si el servidor web está arrancado veremos en la línea de estado algo parecido a : 

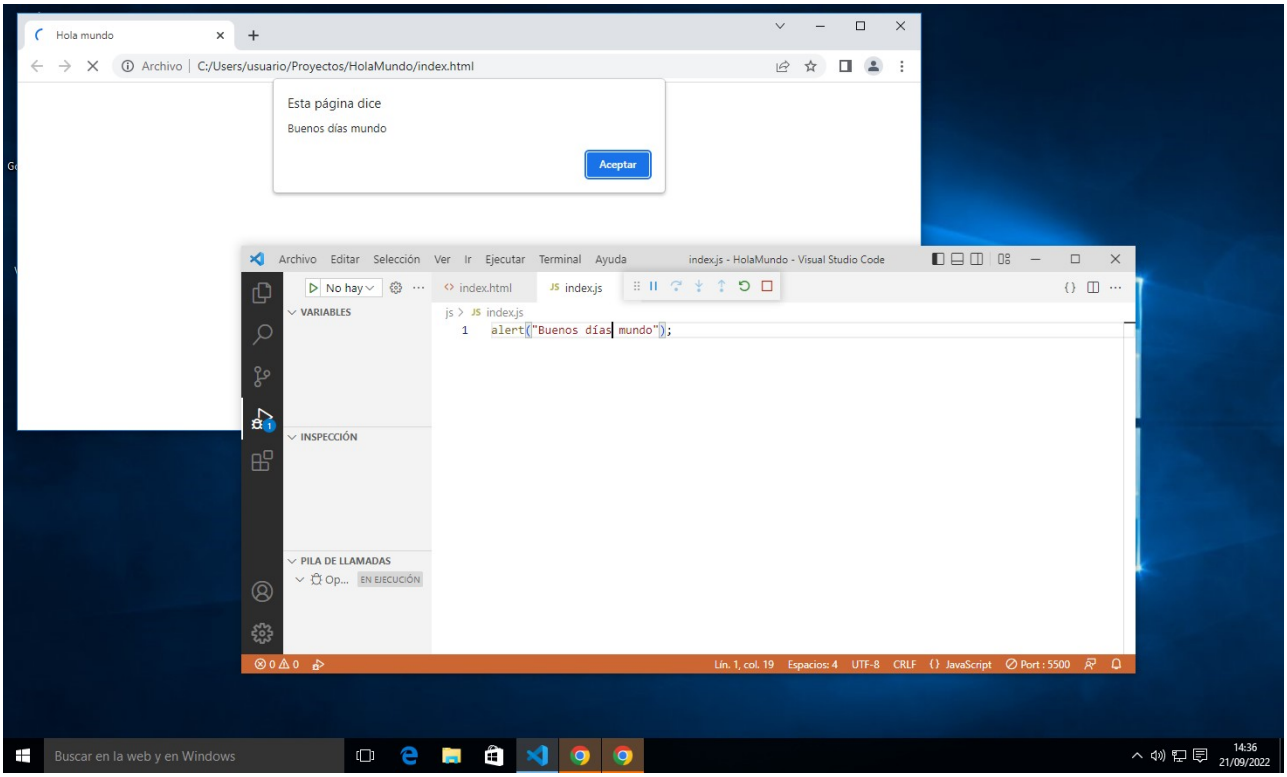
Si queremos volver a ejecutar el código, estando seleccionada la solapa de **index.html** pulsamos **F5** o seleccionamos la opción de menú **Ejecutar → Iniciar depuración**. Si tenemos varios navegadores nos permitirá seleccionar con cual queremos ver la página web desde una lista.



Vemos que ha cambiado el panel izquierdo de **EXPLORADOR** a **VARIABLES** y ha aparecido una barra de herramientas flotante:

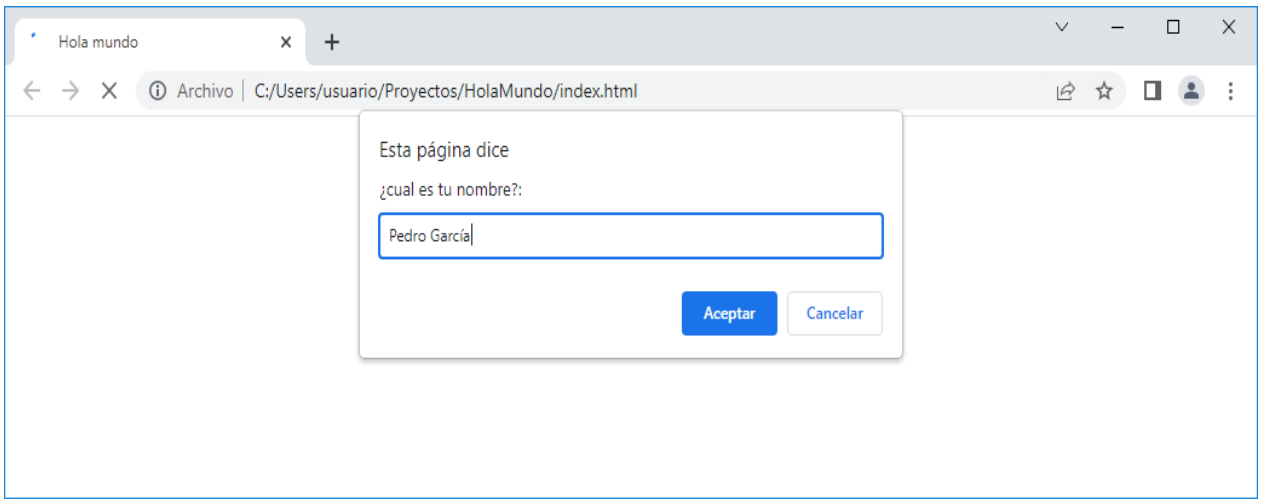
Cuando avancemos más, veremos la utilidad de ambos elementos.

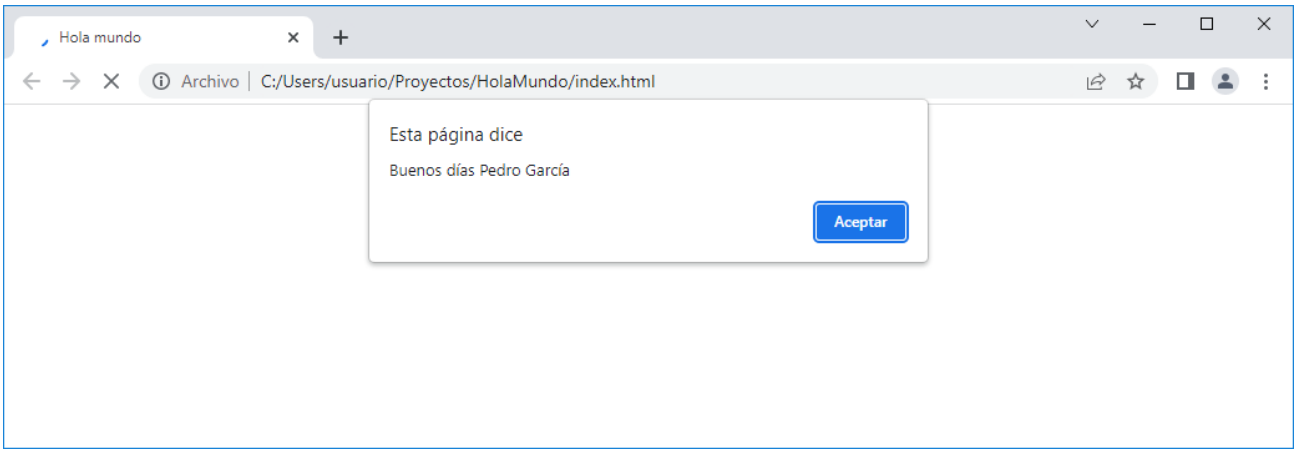
Si en el archivo **index.js** cambiásemos el mensaje a **Buenos días mundo**, no tenemos más que pulsar en  para ver los cambios en la ventana del navegador.



Vamos a hacer ahora una modificación de la anterior: vamos a solicitar al usuario que introduzca su nombre al cargar la página y la aplicación le muestre un mensaje personalizado. Para solicitar el nombre del usuario haremos uso del método **prompt** del objeto **window** ya mencionado. El valor recuperado por el método **prompt** deberá ser almacenado en una variable que vamos a llamar **nombre** y mostraremos su contenido mediante **alert:**

```
var nombre;                          // declaración de la variable
nombre = prompt("¿cual es tu nombre?: ");    // solicitar el nombre
alert("Buenos días "+nombre);              // mostrarlo
```





El par de caracteres **//** indican al intérprete que lo que sigue hasta el final de línea son comentarios que no deben ser traducidos.

Cuando se declara una variable se le puede dar un contenido inicial en la misma sentencia, por ejemplo, **var edad = 21;** En este caso podríamos haber escrito directamente: **var nombre = prompt("¿cual es tu nombre?");** incluso podríamos haber prescindido de la variable **nombre** porque solo va a ser utilizada en el **alert**. Nuestro programa se podría haber escrito:

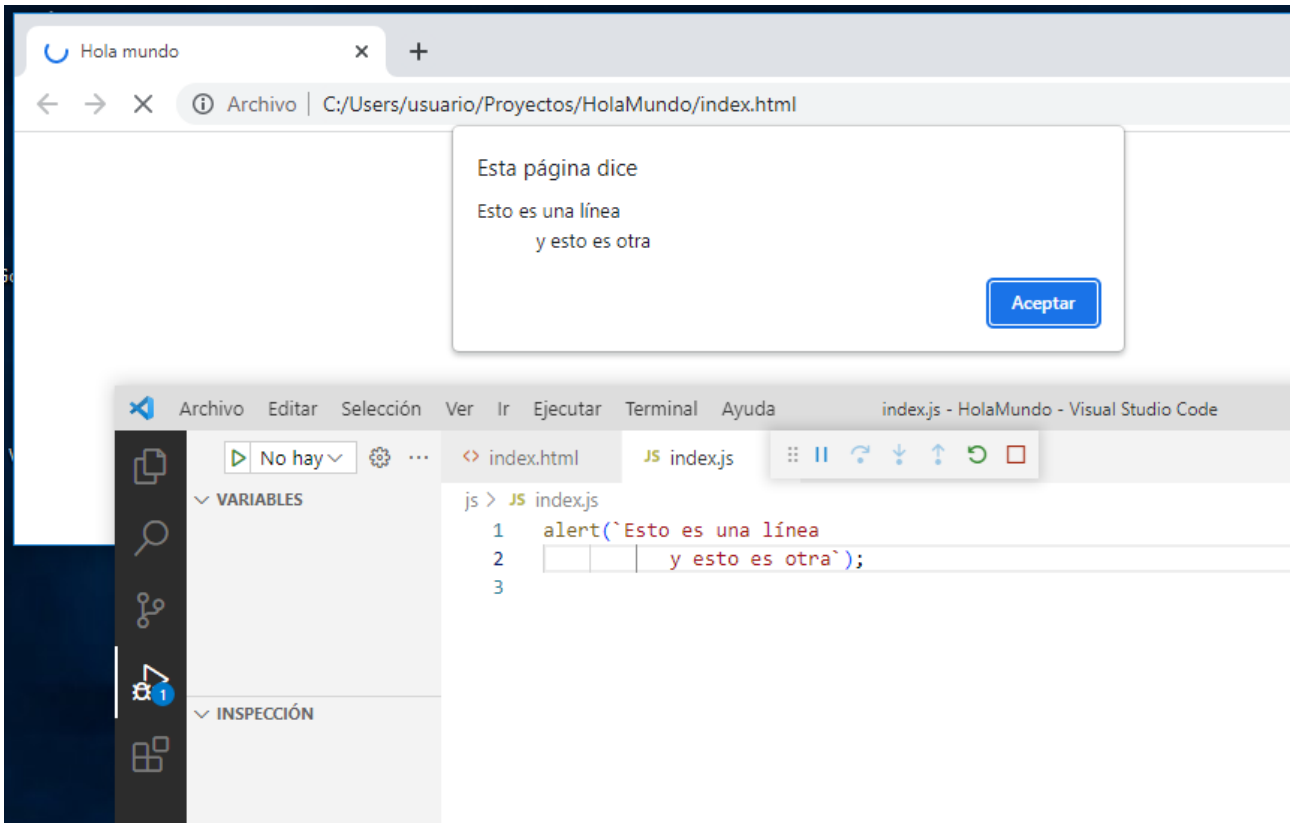
```
alert(("Buenos días "+ prompt("¿cual es tu nombre?: "));
```

Vemos en el parámetro del método **alert** como se ha utilizado el operador **+,** no como operador aritmético, sino como operador para **concatenar** dos cadenas de caracteres. Por un lado **"Buenos días"** y por otro lo que el usuario haya introducido en la petición del **prompt**.

En javaScript hay un nuevo delimitador, además de " y ', para delimitar cadenas de caracteres. Es el carácter de acento grave ` Aunque los otros dos son semejantes en cuanto a funcionalidad, este

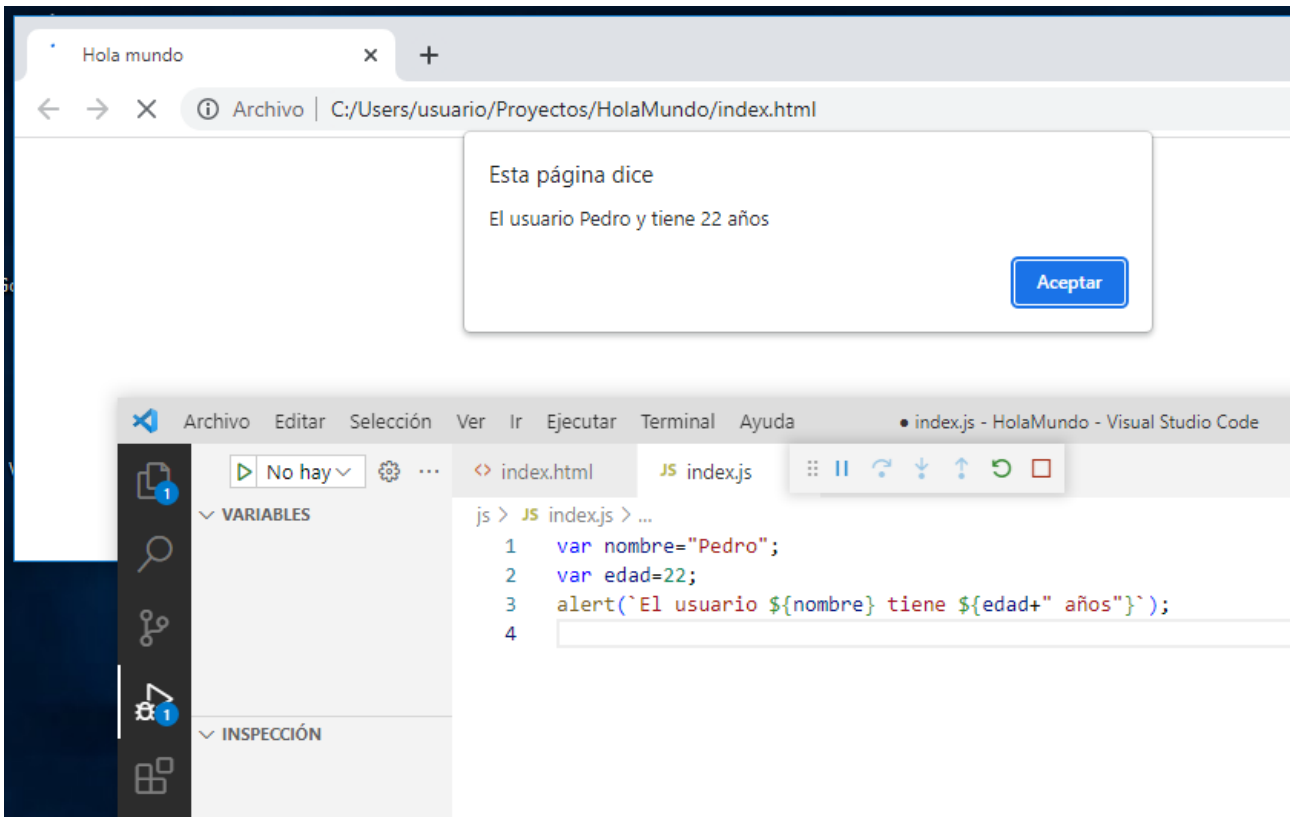
delimitador actúa de forma diferente: permite la escritura de cadenas multilínea sin tener que añadir el carácter **\n** y expande las expresiones que contengan si están encerradas entre llaves precedidas de un carácter **$**. Por ejemplo la siguiente es una cadena **multilínea**

```
alert(`Esto es una línea
      y esto es otra`);
```



En el siguiente se expande una expresión dentro de la cadena:

```
var nombre="Pedro";
var edad=22;
alert(`El usuario ${nombre} tiene ${edad+" años"}`);
```



Entre las llaves se puede poner cualquier expresión javaScript que devuelva una cadena de caracteres.

Este último ejercicio es un claro ejemplo de **programación secuencial**. Una instrucción no se ejecuta hasta que la anterior no ha terminado de ejecutarse y no hay desvíos en la ejecución, siempre se ejecuta la siguiente según la **secuencia** en la que han sido escritas. También se le puede llamar ejecución **síncrona,** aunque este concepto tiene que ver más con operaciones que tardan en realizarse, por ejemplo, la lectura de un archivo desde un dispositivo. En lugar de detener la ejecución hasta que el archivo sea leído, la ejecución continua y cuando se ha recuperado el archivo se lanza un aviso, **evento**, al programa para que haga algo en respuesta. Si el programa espera a que la operación lenta termine,se habla de operación **síncrona.** Si la ejecución continua y se captura el evento de operación lenta terminada, se habla de operación **asíncrona**. Mas adelante se ahondará en estos conceptos. Por ahora solo nos fijamos en que en nuestro programa hemos realizado una ejecución **secuencial**.

Instrucciones condicionales

Normalmente, la ejecución de un programa no se realiza de forma completamente secuencial y dependerá de que en determinados momentos se desee la ejecución de una serie de sentencias, o no, en función de alguna condición. Es aquí donde intervienen las sentencias condicionales y aparece el concepto de **bloque de código**. El ejemplo más básico y universal entre los lenguajes de programación de instrucción condicional es **if.** Su sintaxis es:

if (*condición*) *sentencia1* [else *sentencia2*]

En los manuales técnicos de informática, cuando aparecen referencias a sintaxis se siguen una serie de normas no escritas y no obligatorias pero que todo el mundo suele seguir:

- Las palabras en negrita se han de escribir tal cual. En la sintaxis anterior **if, ()** y **else** se han de escribir exactamente así. No valdría escribir **elSe**, por ejemplo
- Las palabras escritas en cursiva han de ser sustituidas por el elemento particular. Por ejemplo, en la sintaxis anterior *condición* debe ser sustituida por una condición válida por ejemplo **edad > 20**.
- La parte de la sentencia encerrada entre corchetes **[]** es opcional y puede no escribirse. En la sintaxis anterior sería válida una sentencia:

  ```
  if (edad > 18) alert("mayor de edad");
  ```

- Si aparecen puntos suspensivos, significa que el elemento anterior puede repetirse de forma indefinida. Por ejemplo, en la sintaxis siguiente ,*número2* puede repetirse de forma indefinida y por lo tanto la instrucción **Math.max(3, 8, 1, 10);** sería válida

  Math.max(*número1* [, *numero2 ...*]);

- Si aparece un carácter **|** entre dos elementos significa que solo se puede incluir uno de ellos. Por ejemplo:

  new Date(*cadena* | *milisegundos*)

  Indica que o se escribe una *cadena* o se escribe el número de *milisegundos*, pero no ambos

Aunque en la sintaxis de **if** ponga *sentencia1* no se refiere a una única sentencia, puede ser un **bloque de código,** que es un **conjunto de sentencias** que tienen la **categoría de una única sentencia**. Un conjunto de sentencias se dice que es un **bloque de código** si está encerrado entre llaves **{}**.

El siguiente ejemplo de **if** sería correcto:

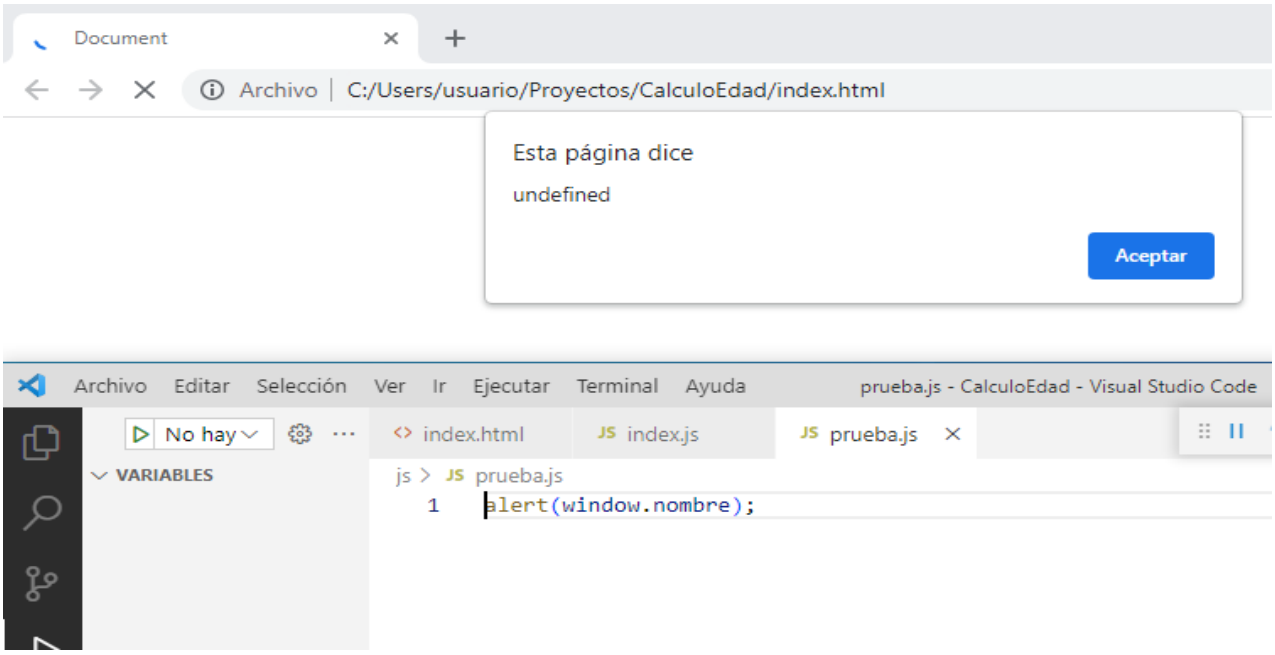
```
if ( salario > 10000)
  {
      retencion = salario *0.2;
      alert(`se le va a practicar una retención de ${retencion} €`);
  }
else
  {
      retencion = salario * 0.01 + 100;
      alert(`poco sueldo, poca retención: ${retencion} €`);
  }
```

Si solo hay una sentencia se puede omitir el uso de las llaves.

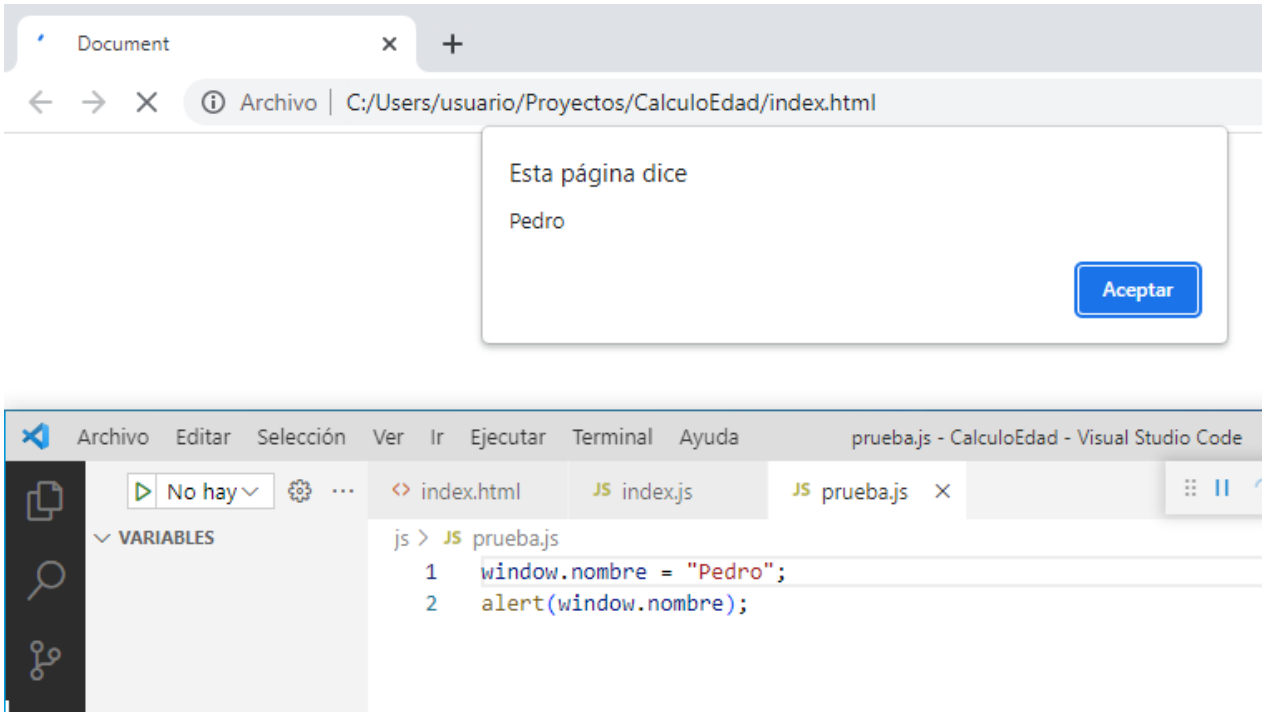
Diferencia entre var y let

Ya vimos que para utilizar una variable debe ser antes declarada haciendo uso de la sentencia **var** o **let** y que cuando viéramos lo que era un bloque de código explicaríamos la diferencia.
Que se haya de declarar un variable antes de su uso puede parecer que no es cierto. Hemos dicho que si hay una referencia a una propiedad o método en la que no aparece el objeto al que

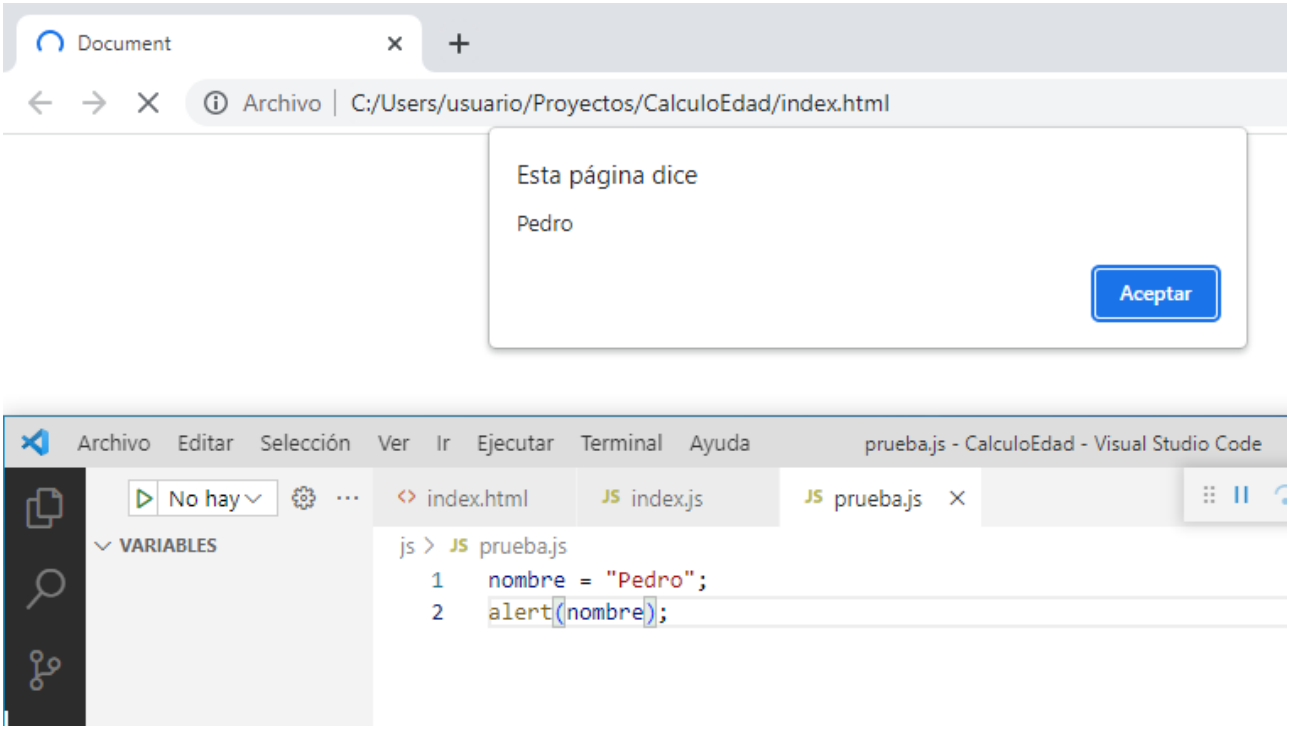
pertenece, se supone que se hace al objeto **window.** En javaScript cuando se hace referencia a una **propiedad** que no existe en el objeto se obtiene un valor **undefined** pero no un error en tiempo de ejecución:



Si se guarda un valor en una propiedad que no existe, la propiedad es creada:

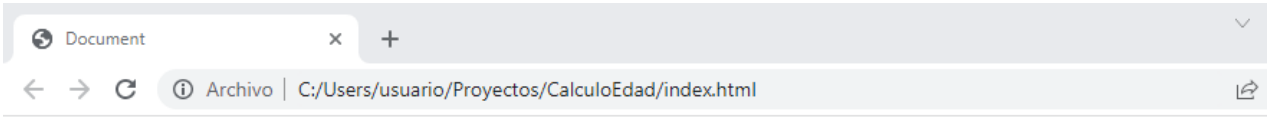


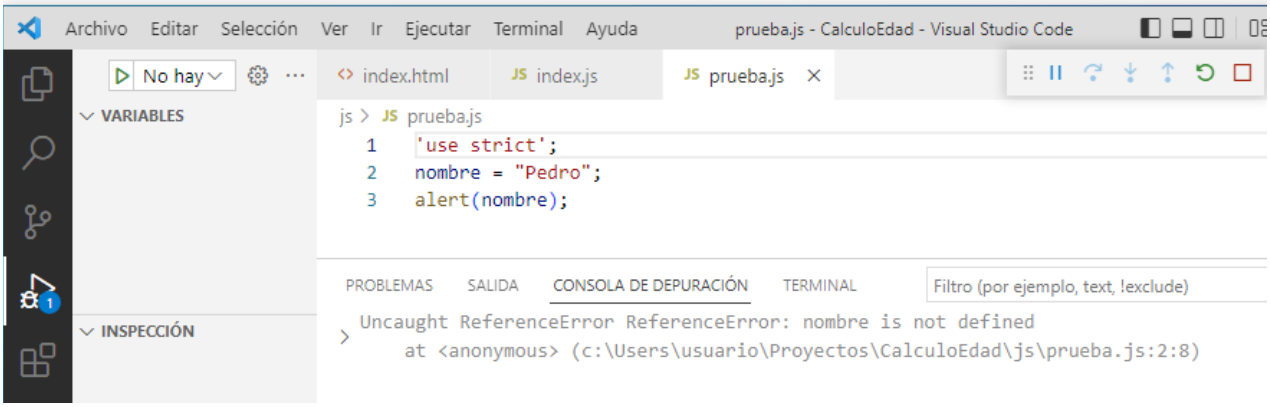
Dado que si no se indica objeto se va a presuponer el objeto **window,** el siguiente código va a funcionar:



Y puede parecer que se ha declarado una variable sin usar ni **var** ni **let.** Lo que se ha hecho es usar una propiedad del objeto **window**.

Si se intenta **recuperar** el valor de una variable que no ha sido declarada se obtiene un error en tiempo de ejecución, y no se presupone que pertenezca al objeto **window.** Si se intenta meter un valor en una variable que no exista se presupondrá el objeto **window** y la variable se creará como propiedad del objeto **window** que recordemos, es global.
Para evitar esta ambigüedad se puede hacer que javaScript trabaje en modo estricto y desencadene errores en tiempo de ejecución cuando se encuentra con variables, propiedades o funciones que no existen. El modo estricto se indica con la cadena **'use strict';** como sentencia separada:

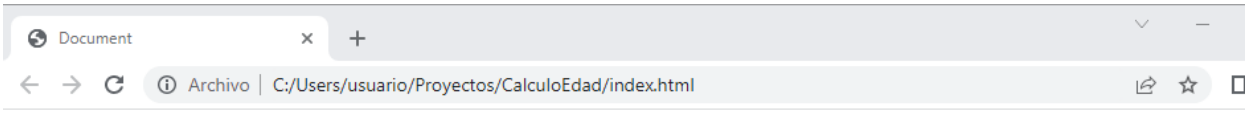


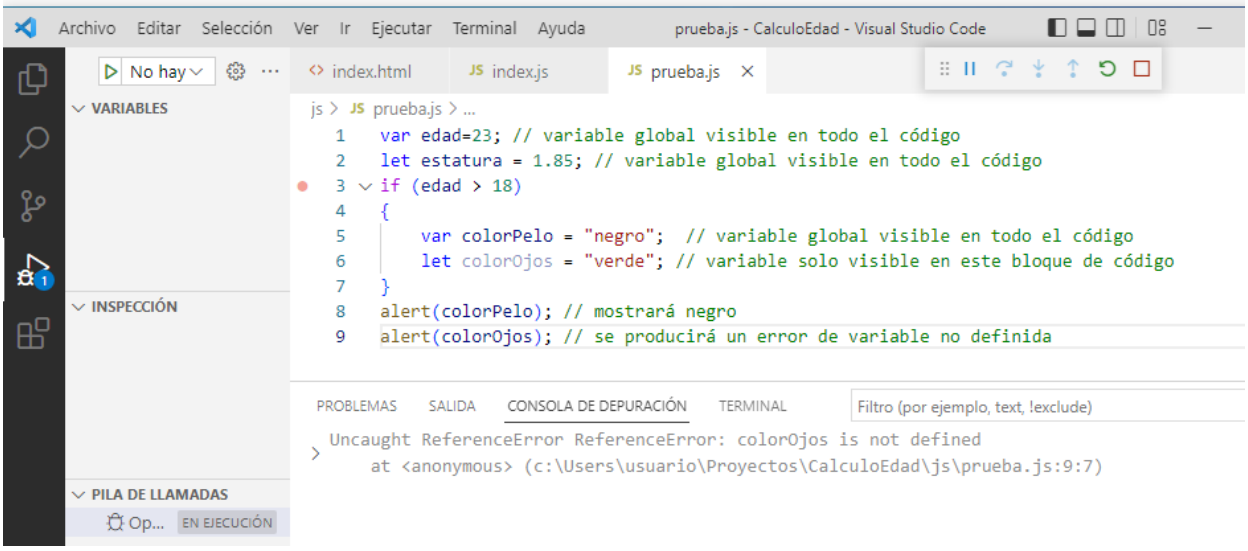


Para entender la diferencia entre declarar una variable con **var** o con **let** debemos hablar de la visibilidad de las variables. Una variable se puede declarar en cualquier lugar. Si una variable es declarada fuera de cualquier bloque de código o función o método, será visible en cualquier lugar del código, con **var** o con **let**, es por tanto una variable **global**. Los objetos del **DOM** son globales, podremos hacer referencia a ellos en cualquier lugar del código.

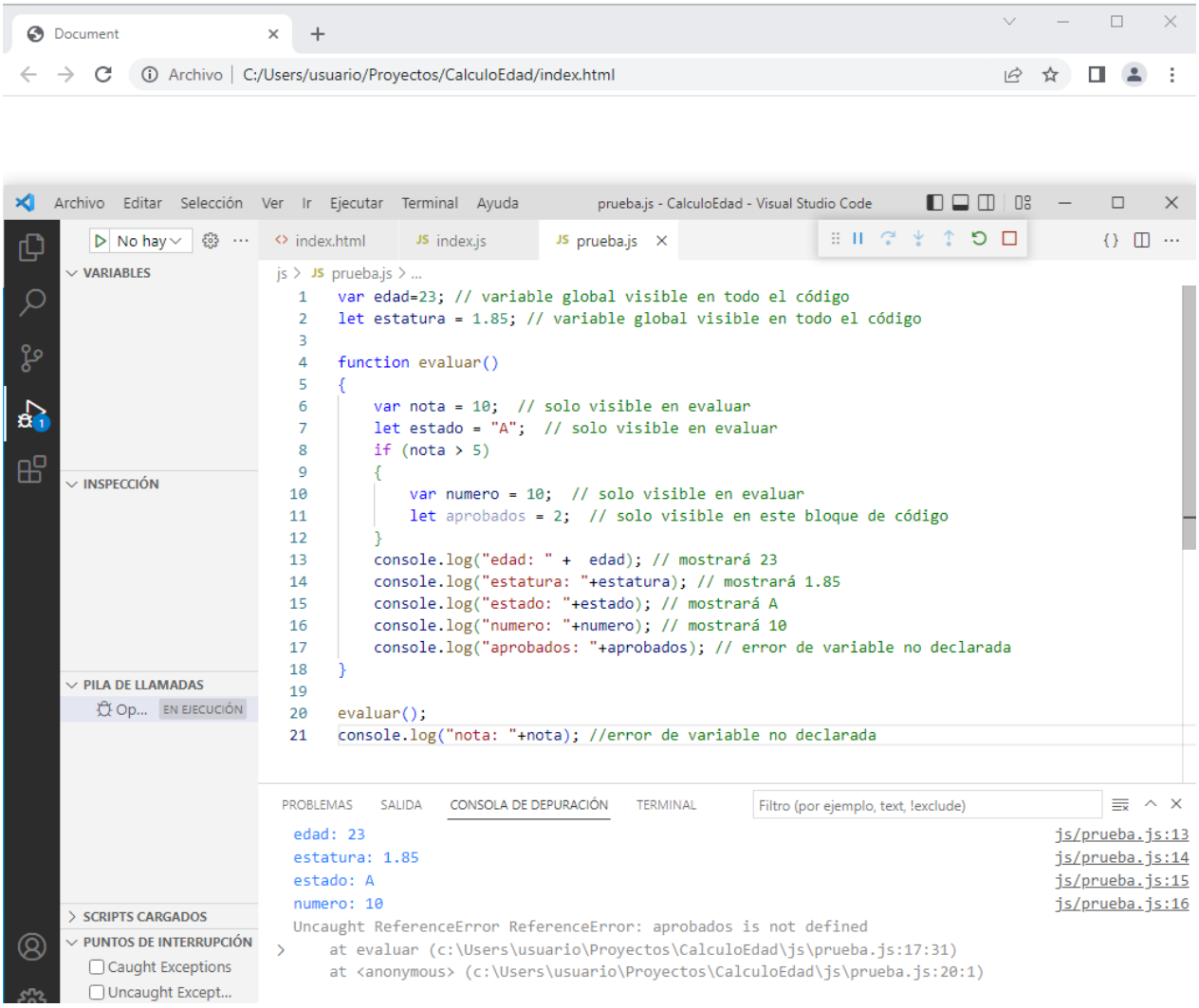
Por contra una variable declarada dentro de una función o método solo será visible o accesible dentro del ámbito de la función o método. Esto es cierto si se declara con **var**. Si una variable se declara dentro de un bloque de código con **let** solo será visible dentro de ese bloque de código. Si dentro de un bloque de código se declara una variable con **var** será visible en el método o función donde se ha declarado y si no lo hubiera sería el ámbito global. En el siguiente ejemplo se ven las diferencias:

```
var edad=23; // variable global visible en todo el código
let estatura = 1.85; // variable global visible en todo el código
if (edad > 18)
{
   var colorPelo = "negro"; // variable global visible en todo el código
   let colorOjos = "verde";  // variable solo visible en este bloque
}
alert(colorPelo); // mostrará negro
alert(colorOjos); // se producirá un error de variable no definida
```





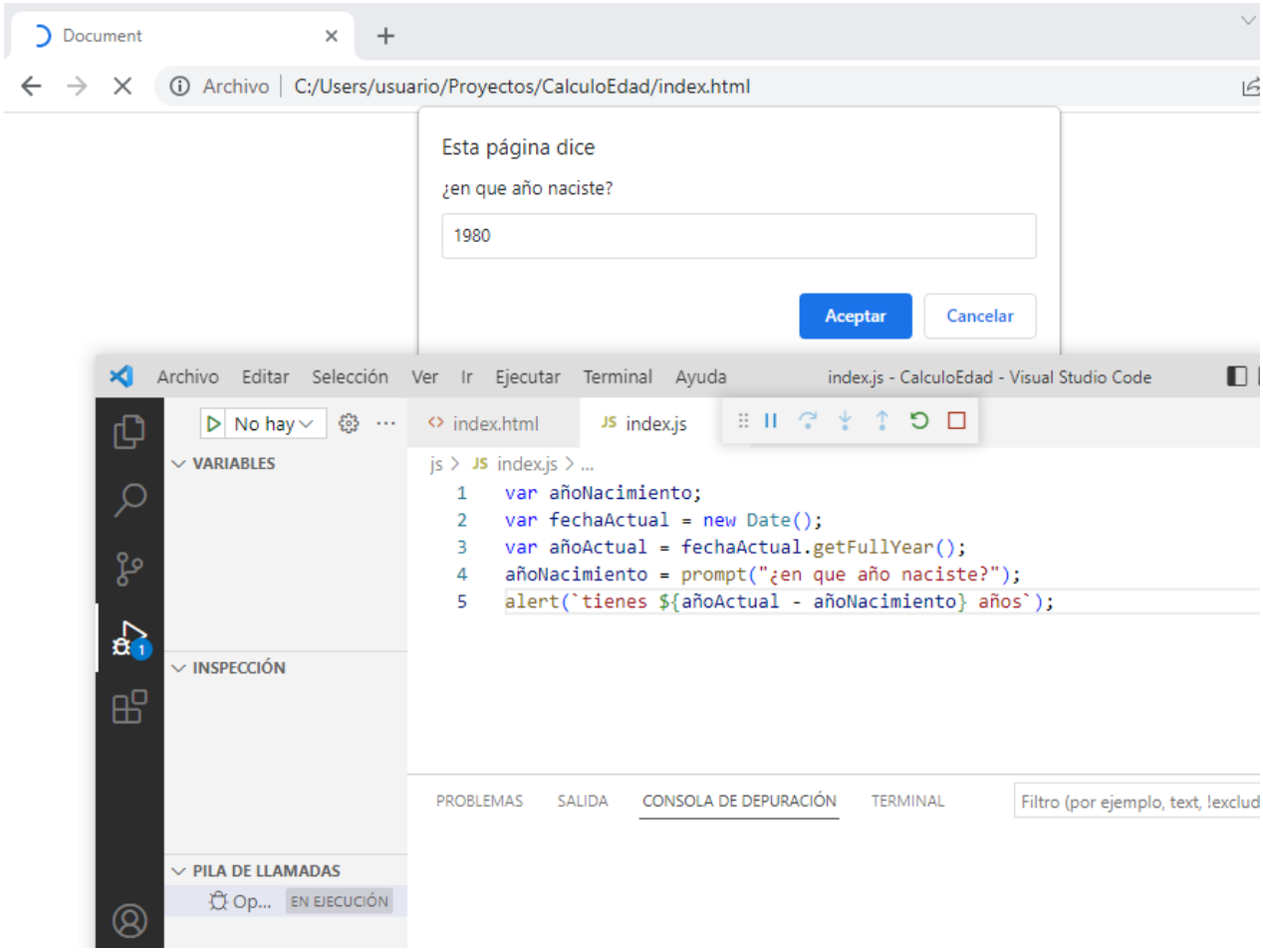

```
var edad=23;            // variable global visible en todo el código
let estatura = 1.85;    // variable global visible en todo el código

function evaluar()
{
   var nota = 10;       // solo visible en evaluar
   let estado = "A";    // solo visible en evaluar
   if (nota > 5)
   {
      var numero = 10;     // solo visible en evaluar
      let aprobados = 2;   // solo visible en este bloque de código
   }
   console.log("edad: " +  edad);                 // mostrará 23
   console.log("estatura: "+estatura);            // mostrará 1.85
   console.log("estado: "+estado);                // mostrará A
   console.log("numero: "+numero);                // mostrará 10
   console.log("aprobados: "+aprobados);     // error de variable no declarada
}

evaluar();
console.log("nota: "+nota);       //error de variable no declarada
```



En el código se ve como se declara una función **evaluar** y como se ejecuta: **evaluar();**. Más adelante trataremos en más profundidad la declaración de funciones y métodos

Vamos a hacer nuestra segunda aplicación. Creamos una nueva aplicación con nombre **CalculoEdad** (creamos una nueva carpeta, un archivo **index.html** con la etiqueta **script**, un archivo **index.js** bajo una carpeta **js …).** La aplicación va a pedir al usuario un **año de nacimiento** y vamos a calcular su **edad**.

```
var añoNacimiento;
var fechaActual = new Date();
var añoActual = fechaActual.getFullYear();
añoNacimiento = prompt("¿en que año naciste?");
alert(`tienes ${añoActual - añoNacimiento} años`);
```



Vemos algún elemento nuevo: **new Date()**. Ha hemos mencionado el concepto de clase y de instancia, pues **Date** es una clase que contiene métodos y propiedades relacionadas con las fechas. Es una clase de javaScript, a diferencia de **window** que pertenece a una clase del **DOM** y por lo tanto creada por el navegador. En javaScript para **WSH** en Windows seguirá existiendo la clase **Date** pero no encontraremos la clase de la que es instancia **window**.

La forma de instanciar una clase en javaScript es haciendo uso del operador **new**. Resumiendo, en la variable **fechaActual** vamos a tener una nueva instancia de la clase **Date**. Al instanciar una clase se le pueden pasar argumentos dentro de los paréntesis a la hora de invocar su construcción con el operador **new.** En este caso no se ha pasado ningún parámetro con lo que la fecha almacenada en la instancia es la actual del momento en que se ejecute. Un objeto de tipo **Date** es capaz de guardar información de una fecha y también de una hora. Por ejemplo, si hubiéramos querido instanciar un **Date** con el 25 de diciembre de 2002 a las 23:05, deberíamos haber codificado:

```
new Date(2002,11,25,23,05);
```

El orden de los parámetros en el constructor es año, mes, día, minutos, segundos, milisegundos, pudiéndose omitir alguno o todos los referidos a la hora. Nótese que se ha escrito **11** para referirnos a diciembre. Los meses van de **0** a **11**. ¿¿¿¿????

También es nuevo: **var añoActual = fechaActual.getFullYear();**

Aunque es nuevo, ya se debería entender. Se está invocando un método con nombre **getFullYear,** sin parámetros puesto que no hay nada entre los parámetros, de la instancia **fechaActual** de la clase **Date.** Este método devuelve el año con cuatro cifras de la fecha almacenada en la instancia. Como en la instancia, al construirla, guardamos la fecha y hora en la que se ejecute la instrucción, tendremos el año actual.

El resultado de la expresión **añoActual - añoNacimiento** es un valor numérico, sin embargo está inmerso en una cadena de caracteres. En las expresiones con mezcla de tipos de datos no habrá problemas si los datos pueden ser convertidos al tipo más amplio, en nuestro caso si el valor numérico puede ser convertido a cadena, que sí se puede. Estas conversiones se pueden forzar invocando métodos de conversión o se hará de forma automática. Los datos numéricos y booleanos que aunque no son objetos se envuelven, **wrap**, en objetos de clase predefinidas que disponen (en realidad todos los objetos) de un método **toString** que es invocado de forma automática cuando se precisa hacer una conversión a cadena. La clase predefinida en la que se envuelve el número es **Number**, la de los valores booleanos es **Boolean** y la de las cadenas es **String**. Por lo tanto, en la expresión **añoActual - añoNacimiento** inmersa en una cadena lo que en realidad se está ejecutando es **(añoActual - añoNacimiento).toString().**

Todas las clases predefinidas o las que cree el programador son herederas del antecesor común **Object** que dispone de el método **toString** que puede ser adaptado en las clases herederas. Sobre estos conceptos de programación con objetos como la herencia, se hablará más adelante.

Dada esta capacidad de javaScript de envolver los datos primitivos en clases, se permite la invocación de los métodos de estas clases desde los propios datos. Por ejemplo, es completamente válido **"Salamanca".toUpperCase()** que devuelve la cadena en mayúsculas.

Las clases en las que se envuelven los datos primitivos se pueden instanciar, aunque no es habitual. Si se instancia una clase de estas no se obtiene lo mismo que si se utiliza el dato sin instanciar. Uno será un objeto y el otro no.

```
var oAnio = new Number(1992);
var intAnio = 1992;
alert(oAnio == intAnio);   // devolverá true, mismo valor numérico
alert(oAnio === intAnio); // devolverá false. No son intrínsecamente lo mismo
```

Parece que ya tenemos nuestro programa funcionando bien, pero ¿que pasa si el usuario en lugar de un año lógico como **2022** escribe una cadena o un valor como **-3.1416** o deja la caja de texto vacía o pulsa cancelar en el **prompt**? Vamos por partes:

- El método **prompt** siempre devuelve una cadena o **null** si se pulsa **Cancel**. n**ull** es un valor especial, nulo, sin tipo de dato asociado. Una variable cuando es definida pero no inicializada tiene un valor **undefined** que es otro valor especial distinto de **null**. Si se quiere definir una variable y dejarla vacía debería ser inicializada a **null:**

  ```
  var dato = null;
  ```

- Si se pulsa **Aceptar** en el prompt sin escribir nada en la caja de texto, la variable que recibe tiene el valor **""** o cadena vacía que no se debe confundir con **null.**

- Cuando en una expresión aritmética aparece un elemento que no es numérico toda la expresión se evalúa al valor especial **NaN** (**N**ot **a N**umber**)** pero no se produce un error en tiempo de ejecución. Si el elemento no numérico de la expresión puede ser convertido de forma automática a un número, la expresión será evaluada de forma correcta. De hecho en la aplicación que estamos desarrollando **añoNacimiento** cuando es recogida desde **pormpt** es guardada en forma de cadena y sin embargo no da ningún error porque en la expresión **añoActual** - **añoNacimiento** se ha hecho una conversión automática de **añoNacimiento** a número porque el operador **-** solo está permitido en expresiones aritméticas. Si en lugar del operador **-** hubiera sido el operador **+ ¿qué habría ocurrido?**

- Para controlar la posibilidad de que se introduzcan datos no lógicos como **3.1416** se debe validar el contenido una vez introducido para limitarlo a valores lógicos como por ejemplo desde **1900** hasta el año actual.

Necesitamos una instrucción que en función del contenido de la variable **añoNacimiento** nos permita mostrar algún mensaje de error si el contenido está fuera de lo aceptable, o mostrar el resultado si todo estaba bien. Esta instrucción es **if**.

```
1.  var añoNacimiento;
2.  var fechaActual = new Date();
3.  var añoActual = fechaActual.getFullYear();
4.  añoNacimiento = prompt("¿en que año naciste?");
5.  if (añoNacimiento != null)
6.  {
7.      if (añoNacimiento == "" ||
8.              isNaN(añoNacimiento) ||
9.              parseInt(añoNacimiento)!= añoNacimiento ||
10.             añoNacimiento < 1900 ||
11.             añoNacimiento > añoActual)
12.         alert("para hacer el cálculo es necesario introducir un año entre 1900 y el año actual");
13.     else
14.         alert(`tienes ${añoActual - añoNacimiento} años`);
15. }
```

Hasta la sentencia **4** es similar al anterior. La línea **5** interroga para ver si se pulsó **Cancelar** en el **prompt**. Si así fue la ejecución saltaría tras las llaves a la parte **else,** que no hay, y por lo tanto la ejecución terminaría ahí.

En la línea **7** se interroga para saber si lo introducido se corresponde con un año entre **1900** y el contenido de **añoActual** que contiene el año de la fecha en que se ejecuta el código:

- **añoNacimiento == ""** si no se introdujo nada en la caja de texto y se pulsó **Aceptar**. Esta condición sobraría porque la siguiente condición también lo detectaría.
- **isNaN(añoNacimiento)** La función **isNaN** indica si el argumento pasado no es numérico. Una función es similar a un método pero no está asociada a ningún objeto. Una función o método es el nombre que referencia a un conjunto de instrucciones que son ejecutadas cuando se invoca el nombre de la función o método, es decir, es un bloque de código con nombre. Más adelante se verá en detalle el concepto de función y método.

- **parseInt(añoNacimiento)!= añoNacimiento .** La función **parseInt** devuelve el valor entero de lo pasado como argumento. Si recibe una cadena la trasforma a número entero, si recibe un número con decimales los quita. Resumiendo, la condición dice que si el **añoNacimiento,** que ya sabemos que es numérico**,** después de haber sido trasformado en entero sigue siendo el mismo valor es que era un entero.
- **añoNacimiento < 1900 || añoNacimiento > añoActual**. Comprueba si el año está entre los límites

En javaScript, cuando en una expresión lógica se puede ya deducir el resultado, antes de terminar de evaluar todas las condiciones, la evaluación termina devolviendo este valor. En la expresión compuesta anterior hay varios **OR** encadenados. La tabla de verdad de **OR** dice que si uno de los dos operandos es **true** el resultado es **true**. En la expresión anterior si por ejemplo la expresión **añoNacimiento == ""** es **true**, no hace falta seguir evaluando, porque independientemente del resultado del resto de la expresión el resultado será **true**. Se dice que javaScript hace **short circuit evaluation** a la hora de evaluar expresiones. Es importante saber esto porque podemos aprovecharnos de ello para evitar que hacer evaluaciones separadas cuando una de ellas puede provocar un error en tiempo de ejecución. Por ejemplo, si en una operación de división el divisor es **0,** el resultado es infinito y provoca un error de overflow en tiempo de ejecución. La siguiente evaluación no producirá un error si la variable **a** contiene un **0:**

```
if (a !=0 && ((100/a) >15))
```

Pero la siguiente si:

```
if ( ((100/a) >15) && a !=0)
```

Nótese que en la anterior expresión, como ya sabemos en que orden va a ser evaluada, podíamos haber prescindido de los paréntesis: **if (100 / a > 15 && a != 0)**

La aplicación anterior está mal enunciada. Con solo el año de nacimiento no podemos saber la edad de una persona, necesitamos la fecha completa. Vamos a modificarla para pedir al usuario una fecha completa y así avanzar un poco más en la utilización de métodos de la clase **Date** y de la clase **String**, además de ver más ejemplos de la sentencia **if**.

```
var edad=0;
var añoNacimiento=0;
var mesNacimiento=0;
var díaNacimiento=0;
var fechaActual = new Date();
var añoActual = fechaActual.getFullYear();
var mesActual = fechaActual.getMonth()+1;  // porque en javascript el mes va de 0 a 11
var díaActual = fechaActual.getDate();
añoNacimiento = prompt("¿en que año naciste?");
if (añoNacimiento != null) {
   if (añoNacimiento == "" ||
          isNaN(añoNacimiento) ||
          parseInt(añoNacimiento) != añoNacimiento ||
          añoNacimiento < 1900 ||
          añoNacimiento > añoActual)
      alert("para hacer el cálculo es necesario introducir un año entre 1900 y el año actual");
   else {
```

```
        mesNacimiento = prompt("¿en que mes naciste?");
        if (mesNacimiento != null) {
           if (mesNacimiento == "" ||
                    isNaN(mesNacimiento) ||
                    parseInt(mesNacimiento) != mesNacimiento ||
                    mesNacimiento < 1 ||
                    mesNacimiento > 12)
              alert("para hacer el cálculo es necesario introducir un mes entre 1 y el 12");
           else {
              díaNacimiento = prompt("¿en que día naciste?");
              if (díaNacimiento != null) {
                 if (díaNacimiento == "" ||
                          isNaN(díaNacimiento) ||
                          parseInt(díaNacimiento) != díaNacimiento ||
                          díaNacimiento < 1 ||
                          díaNacimiento > 31)
                    alert("para hacer el cálculo es necesario introducir un día entre 1 y el 31");
                 else {
                    // año mes y día están entre límites
                    edad = añoActual - añoNacimiento;
                      // si mes y día de la fecha de nacimiento es mayor que mes y
                      // día de la fecha actual hay que corregir en 1
                    if (mesNacimiento > mesActual ||
                          (mesNacimiento == mesActual &&
                              díaNacimiento > díaActual))
                       edad--;
                    alert(`tienes ${edad} años de edad`);
                    if (mesNacimiento == mesActual &&
                              díaNacimiento == díaActual)
                       alert("Felíz cumpleaños");
                 }
              }
           }
        }
     }
  }
```

A destacar, los métodos **getMonth** que devuelve el mes entre 0 y 11, por eso hay que corregir sumando 1 para que coincida con la idea de mes que tenemos los humanos, y **getDate** que devuelve el día del mes. También a destacar la anidación de **if** que se visualiza bien si el código está indentado de forma correcta, lo contrario dificultaría muchísimo su interpretación, por ejemplo el mismo código:

```
var edad=0;
var añoNacimiento=0;
var mesNacimiento=0;
var díaNacimiento=0;
var fechaActual = new Date();
var añoActual = fechaActual.getFullYear();
var mesActual = fechaActual.getMonth() + 1;  // porque en javascript el mes va de 0 a 11
var díaActual = fechaActual.getDate();
añoNacimiento = prompt("¿en que año naciste?");
if (añoNacimiento != null) {
if (añoNacimiento == "" ||
isNaN(añoNacimiento) ||
parseInt(añoNacimiento) != añoNacimiento ||
añoNacimiento < 1900 ||
añoNacimiento > añoActual)
```

```
alert("para hacer el cálculo es necesario introducir un año entre 1900 y el año actual");
else {
mesNacimiento = prompt("¿en que mes naciste?");
if (mesNacimiento != null) {
if (mesNacimiento == "" ||
isNaN(mesNacimiento) ||
parseInt(mesNacimiento) != mesNacimiento ||
mesNacimiento < 1 ||
mesNacimiento > 12)
alert("para hacer el cálculo es necesario introducir un mes entre 1 y el 12");
else {
díaNacimiento = prompt("¿en que día naciste?");
if (díaNacimiento != null) {
if (díaNacimiento == "" ||
isNaN(díaNacimiento) ||
parseInt(díaNacimiento) != díaNacimiento ||
díaNacimiento < 1 ||
díaNacimiento > 31)
alert("para hacer el cálculo es necesario introducir un día entre 1 y el 31");
else {
// año mes y día están entre límites
edad = añoActual - añoNacimiento;
// si mes y día de la fecha de nacimiento es mayor que mes y día de la fecha actual hay que corregir en
1
if (mesNacimiento > mesActual || (mesNacimiento == mesActual && díaNacimiento > díaActual))
   edad--;
alert(`tienes ${edad} años de edad`);
if (mesNacimiento == mesActual && díaNacimiento == díaActual)
   alert("Felíz cumpleaños");
}
}
}
}
}
}
```

Sería prácticamente indescifrable, aunque se ejecutaría bien igualmente. Como el código javaScript se puede ver cuando vemos una página de cualquier sitio, hay un técnica que se dice de **ofuscación** que consiste en dificultar todo lo posible la interpretación del código eliminando cualquier tipo de sangrado, dejando todo el código en una línea y sustituyendo los nombres de variables y funciones o métodos por algo sin sentido, por ejemplo en lugar de llamar a la variable **añoNacimiento** la llamamos **b35**. Como la ofuscación es también para el programador lo que se hace es que el código que utiliza el programador no está ofuscado y a la hora de subir el código a producción se pasa por una herramienta de ofuscación de código.

El código anterior tiene todavía algún problema,por ejemplo el usuario podría poner como fecha de su nacimiento el 31 de febrero de algún año, o el 29 de febrero de algún año que no sea bisiesto, o el 31 de abril .... El código debería detectarlo y avisar.

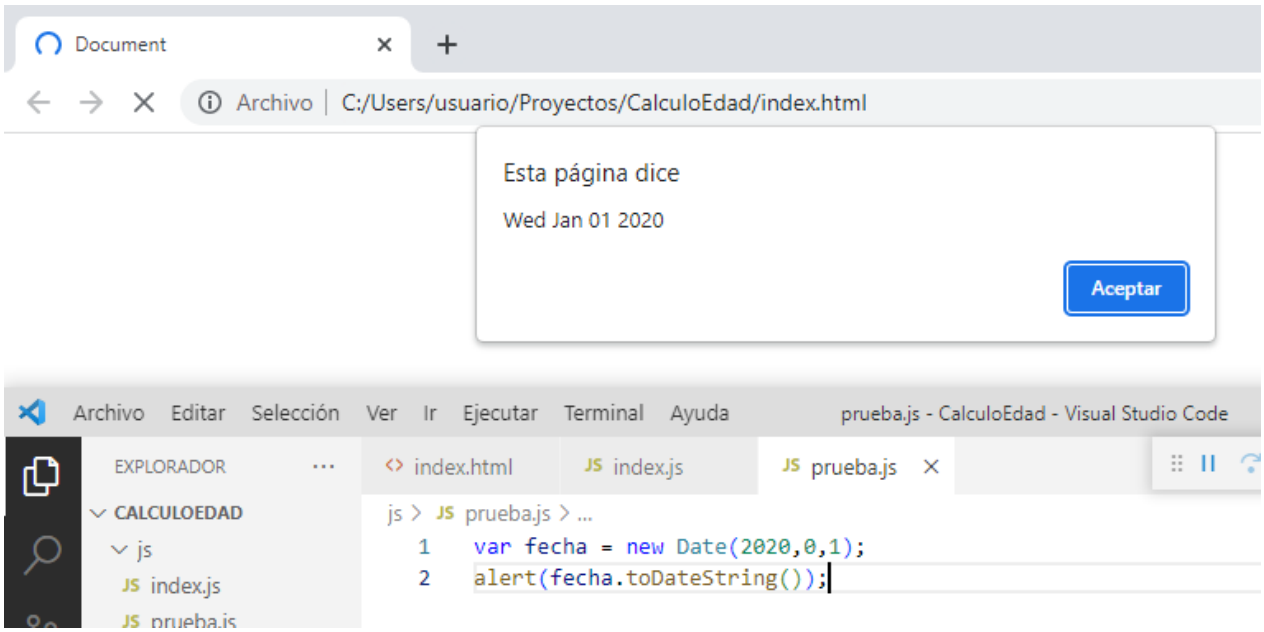
La clase **Date** permite manejar una fecha y una hora y dispone de métodos para manejar ambos datos. Dispone de métodos para recuperar cada componente, se llaman métodos **getter**. Así tenemos el método **getMonth** que nos permite recuperar el mes de la fecha contenida. También dispone de métodos para modificar cada componente, se llaman métodos **setter**. Así tenemos un método llamado **setMonth** que permite poner el mes de la fecha contenida. En

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Date tenemos una referencia completa de todos sus métodos.

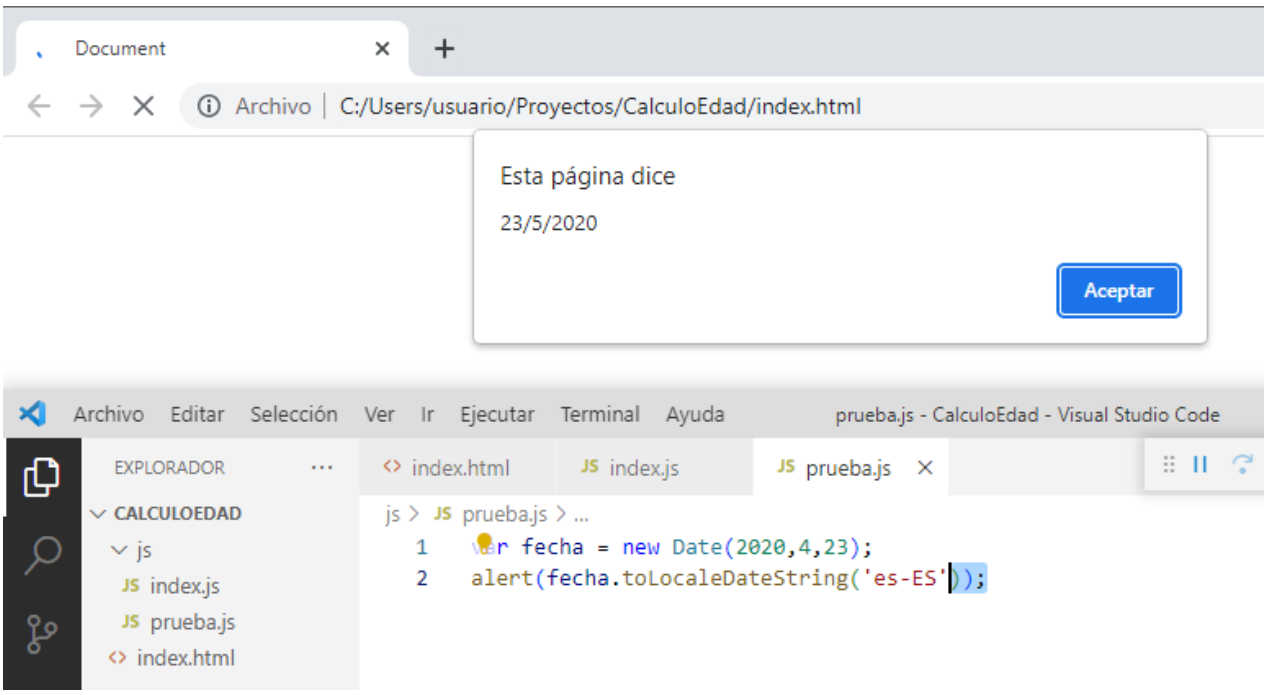
Es curiosa la forma en la que el objeto **Date** maneja datos en apariencia erróneos. Por ejemplo, si almacenamos en una fecha el **32 de enero** de **2002** en lugar de dar error lo que hace es guardar el **1 de febrero de 2002**. Esto que puede parecer un mal funcionamiento en realidad es una ventaja, por ejemplo, si queremos saber que fecha es 40 días después del 1 de enero de 2020, no tenemos más que sumar 40 al día de la fecha y guardarlo como nuevo día:

```
var fecha = new Date(2020,0,1);
fecha.setDate(fecha.getDate()+40);
alert(fecha);
```

Aparte de **getters** y **setters** la clase **Date** dispone de métodos para obtener la fecha y hora como cadena y como cadena de una de la partes fecha u hora, o con formato. Para distintas regiones Por ejemplo, Si solo queremos obtener la fecha como cadena usamos **toDateString()**

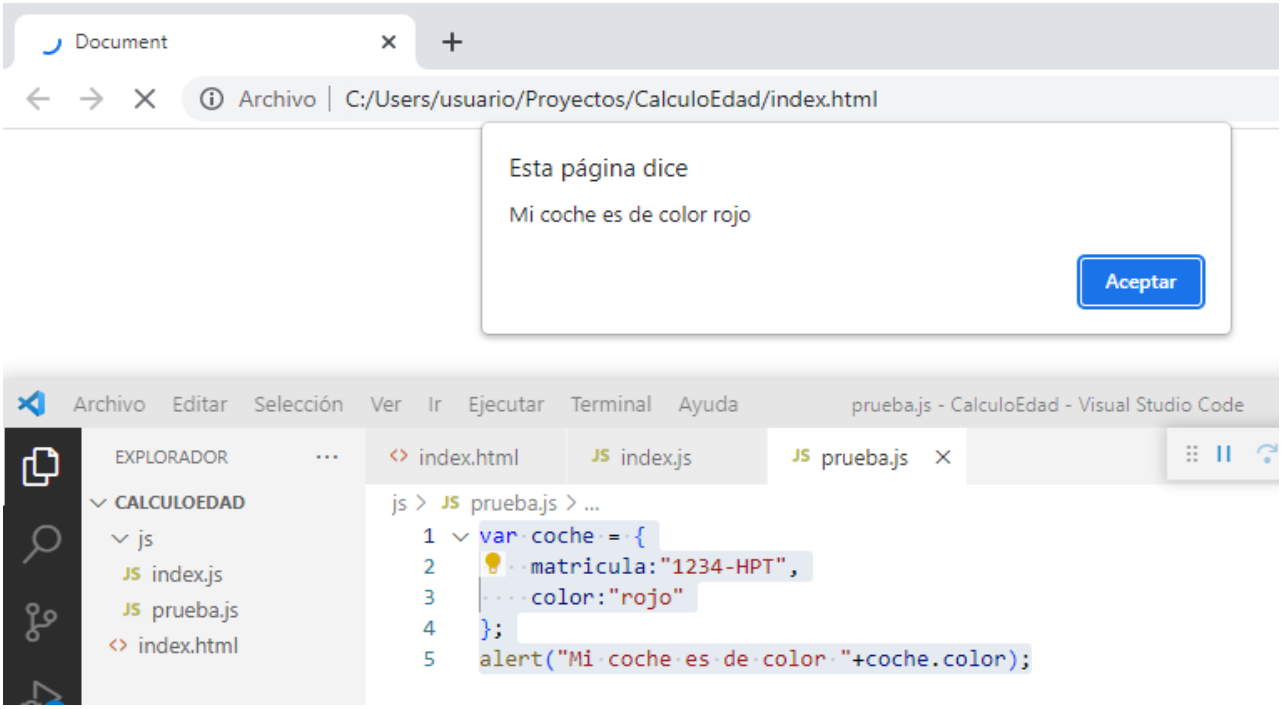


Como se ve, siempre en formato inglés. Si quisieramos la fecha en formato español usaremos el método **toLocaleDateString('es-ES')**



Vemos que ha cambiado el formato, que aparece en día, mes y año como en España, pero solo aparecen números y slash. El método **toLocaleDateString** puede tener un segundo argumento que permite indicar el formato de la fecha en la salida, por ejemplo, para que aparezca el nombre del mes o el nombre del día. Este segundo parámetro debe ser un objeto con unas propiedades con nombre prefijado. Todavía no sabemos como definir clases nuestras, aunque ya sabemos instanciar con el operador **new**. Se puede crear una instancia de una clase sin nombre, **anónima**, simplemente enumerando las propiedades y definiendo los métodos como si fuera un bloque de código. En el siguiente ejemplo se ve que la variable **coche** es una instancia de un clase anónima con una propiedad **matricula** y una propiedad **color.**

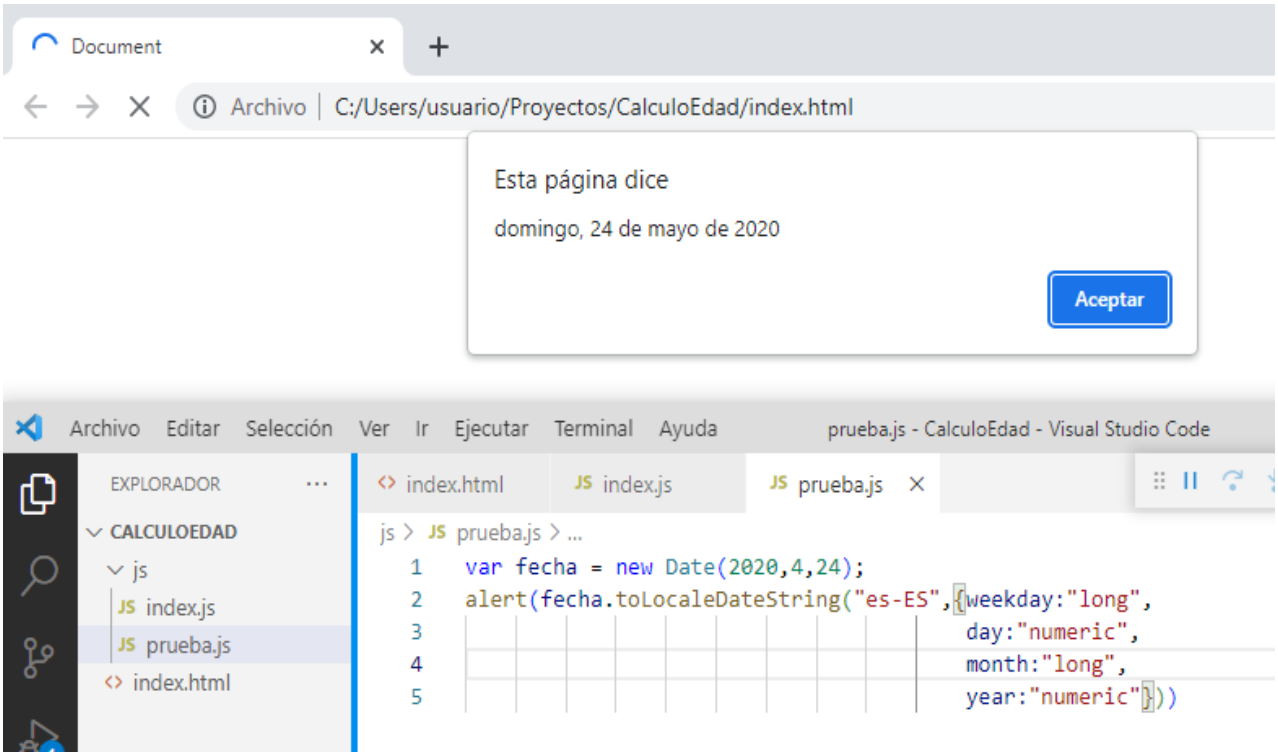
```
var coche = {
   matricula:"1234-HPT",
   color:"rojo"
};
alert("Mi coche es de color "+coche.color);
```



Volviendo al método **toLocaleDateString**, el segundo parámetro debe ser la instancia de un objeto que tenga una o varias de las siguientes: **weekday, year, month, day**. Y las propiedades pueden tener alguno de los siguientes valores:

propiedad	valor	
weekday	**long**	nombre largo del día
	short	nombre abreviado del día
	narrow	inicial del nombre del día
year	**numeric**	cuatro cifras
	2-digit	dos cifras
month	**long**	nombre largo del mes
	short	nombre abreviado del mes
	narrow	inicial del nombre del mes
	numeric	cuatro cifras
	2-digit	dos cifras
day	**numeric**	cuatro cifras
	2-digit	dos cifras

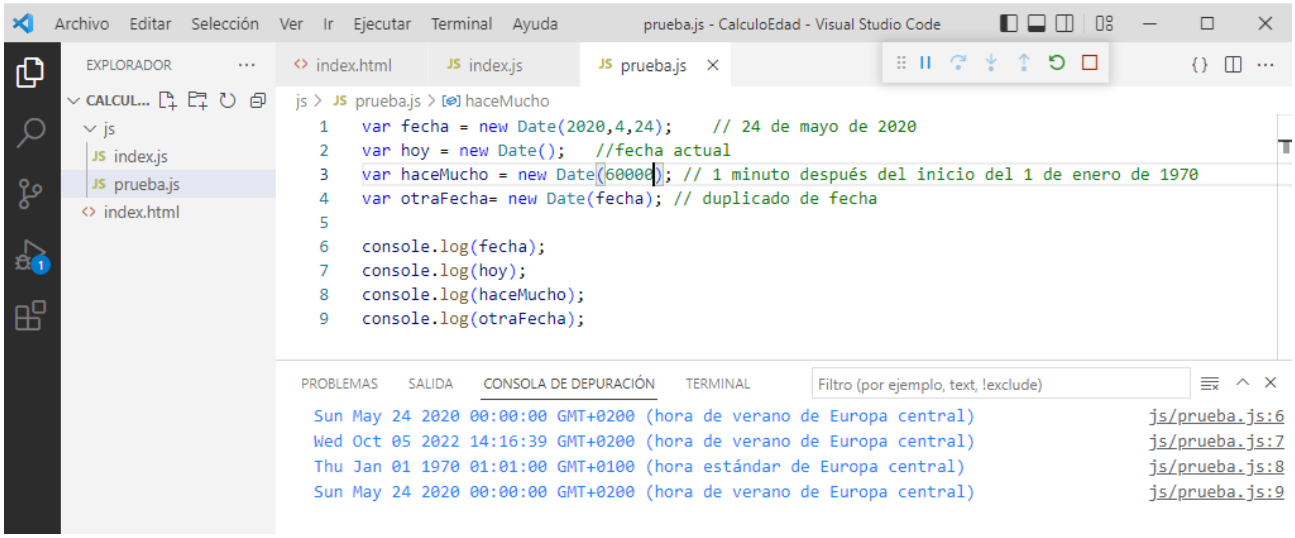
El siguiente ejemplo muestra la fecha con el nombre del día, el número del día, el nombre del mes y el año:



Internamente lo que almacena un objeto de tipo **Date** es el número de milisegundos transcurridos desde el uno de enero de 1970 que es el origen de los tiempo Unix. Almacenará un valor negativo para las fechas anteriores. El método **getTime** devuelve ese número de milisegundos.

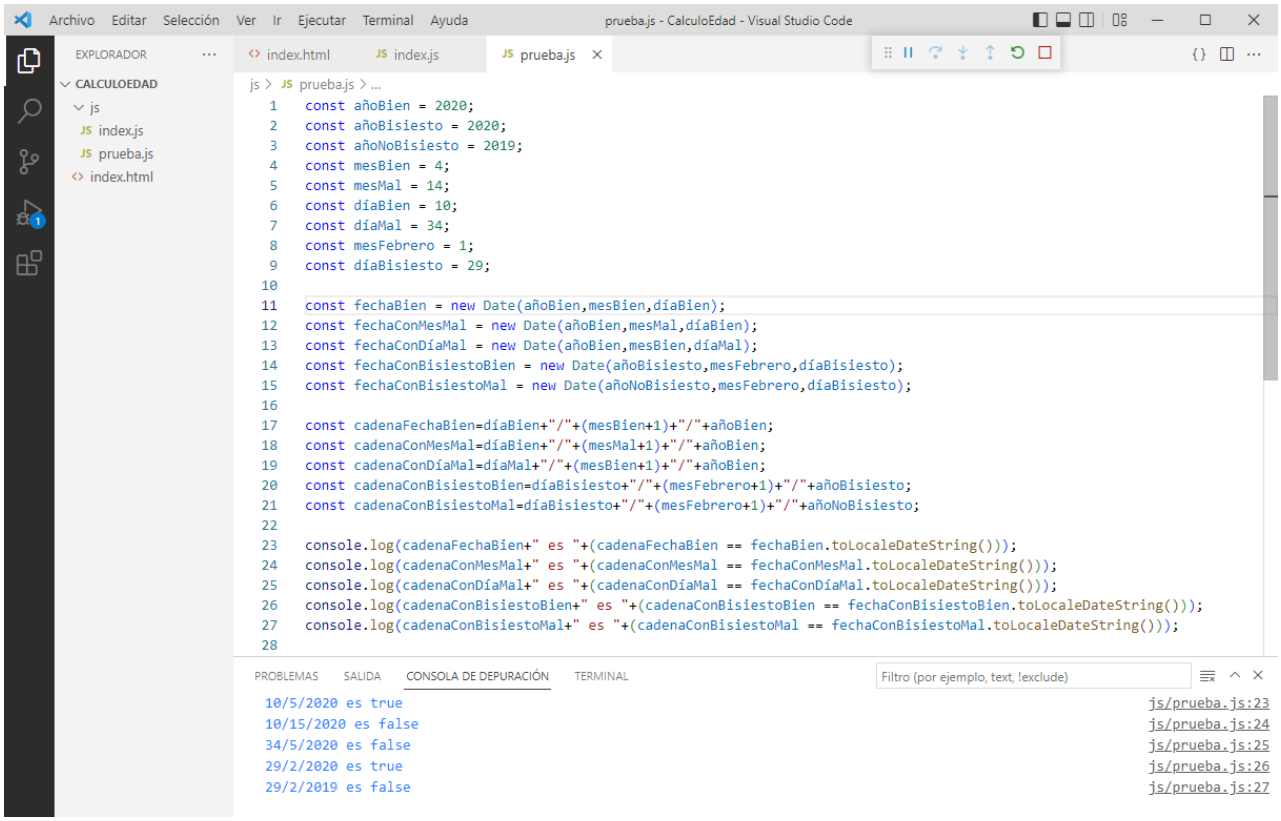
Cuando se invoca la constructor de la clase **Date** ya hemos visto que se le puede pasar un año, mes, día y hora, minutos, segundos y milisegundos como parámetros. También que si se invoca sin argumentos guarda la fecha y hora en la que se invoca. El constructor también puede recibir un único valor entero que representa los milisegundos de la fecha y hora, y también puede recibir otra instancia de la clase **Date**, con lo que tendremos un duplicado:

```
var fecha = new Date(2020,4,24);      // 24 de mayo de 2020
var hoy = new Date();                 //fecha actual
var haceMucho = new Date(60000);      // 1 minuto después del inicio del 1 de enero de 1970
var otraFecha= new Date(fecha);       // duplicado de fecha

console.log(fecha);
console.log(hoy);
console.log(haceMucho);
console.log(otraFecha);
```



Podemos continuar con nuestro programa. ¿Como podemos saber si unos datos de año, mes y día son correctos? Fácil, instanciamos una fecha con los datos, si al recuperarla tenemos la misma fecha es que era correcta, sino alguno de los datos cambia era incorrecta. En el ejemplo siguiente se ve:

```
const añoBien = 2020;
const añoBisiesto = 2020;
const añoNoBisiesto = 2019;
const mesBien = 4;
const mesMal = 14;
const díaBien = 10;
const díaMal = 34;
const mesFebrero = 1;
const díaBisiesto = 29;

const fechaBien = new Date(añoBien,mesBien,díaBien);
const fechaConMesMal = new Date(añoBien,mesMal,díaBien);
const fechaConDíaMal = new Date(añoBien,mesBien,díaMal);
const fechaConBisiestoBien = new Date(añoBisiesto,mesFebrero,díaBisiesto);
const fechaConBisiestoMal = new Date(añoNoBisiesto,mesFebrero,díaBisiesto);

const cadenaFechaBien=díaBien+"/"+(mesBien+1)+"/"+añoBien;
const cadenaConMesMal=díaBien+"/"+(mesMal+1)+"/"+añoBien;
const cadenaConDíaMal=díaMal+"/"+(mesBien+1)+"/"+añoBien;
const cadenaConBisiestoBien=díaBisiesto+"/"+(mesFebrero+1)+"/"+añoBisiesto;
const cadenaConBisiestoMal=díaBisiesto+"/"+(mesFebrero+1)+"/"+añoNoBisiesto;

console.log(cadenaFechaBien+" es "+(cadenaFechaBien == fechaBien.toLocaleDateString()));
console.log(cadenaConMesMal+" es "+(cadenaConMesMal ==
fechaConMesMal.toLocaleDateString()));
console.log(cadenaConDíaMal+" es "+(cadenaConDíaMal == fechaConDíaMal.toLocaleDateString()));
console.log(cadenaConBisiestoBien+" es "+
    (cadenaConBisiestoBien == fechaConBisiestoBien.toLocaleDateString()));
console.log(cadenaConBisiestoMal+" es "+(cadenaConBisiestoMal ==
fechaConBisiestoMal.toLocaleDateString()));
```



Nótese el uso de **const** ya que las variables en ningún momento cambian su valor. En casi todos los ejemplos anteriores se podría también haber usado.

Ya podemos modificar el código de nuestro programa para detectar si el año, mes y día pedidos al usuario son válidos:

```
else {
        díaNacimiento = prompt("¿en que día naciste?");
        if (díaNacimiento != null) {
          if (díaNacimiento == "" ||
            isNaN(díaNacimiento) ||
            parseInt(díaNacimiento) != díaNacimiento ||
            díaNacimiento < 1 ||
            (díaNacimiento+"/"+mesNacimiento+"/"+añoNacimiento) !=
           (new Date(añoNacimiento,mesNacimiento -1,díaNacimiento)).toLocaleDateString("es-ES"))
                  {
                      let díasMes=
                      (new Date(añoNacimiento,mesNacimiento,0)).getDate();
                       alert(`para hacer el cálculo es necesario introducir un día entre 1 y ${díasMes}`);
                  }
            else {
              // año mes y día están entre límites
              edad = añoActual - añoNacimiento;
                // si mes y día de la fecha de nacimiento es mayor que mes y día de la fecha
                // actual hay que corregir en 1
              if (mesNacimiento > mesActual ||
                        (mesNacimiento == mesActual && díaNacimiento > díaActual))
                 edad--;
              alert(`tienes ${edad} años de edad`);
              if (mesNacimiento == mesActual && díaNacimiento == díaActual)
```

```
                alert("Felíz cumpleaños");
            }
        }
    }
```

El único código cambiado es el de la comprobación del día ya que los límites del año y del mes son valores constantes que no dependen de la fecha. En cuanto al día depende del mes y de si es bisiesto el año. Gracias a la particularidad del funcionamiento del objeto **Date** si guardamos en un fecha el día anterior al primero de un mes obtendremos el número del último día del mes anterior: **(new Date(añoNacimiento,mesNacimiento,0)).getDate();** . El primer día de un mes es el **1** siempre, por lo que el día **0** es el último del mes anterior y se corresponderá con el número de días del mes anterior. En la fecha estamos poniendo **mesNacimiento**; como los meses en javaScript van de 0 a 11, **mesNacimiento** sin restar **1** se corresponde con el mes siguiente. Por ejemplo, si el usuario ha puesto mes **2** estaría poniendo **febrero** pero en javaScript el mes **2** es **marzo** por lo que **new Date(añoNacimiento,2,0))** se refiere al día anterior al primero de **marzo**, o sea, el último día de **febrero**. El objeto **Date** sabe cuantos días tiene cada mes y si el año es bisiesto o no ahorrándonos cálculos.

Nótese que **no** se ha declarado una variable para obtener ese dato de fecha, **(new Date(añoNacimiento,mesNacimiento,0)),** para el cálculo de días del mes. Se podría haber hecho pero no es necesario, podemos decir que hemos utilizado una instancia anónima que solo puede ser utilizada una vez. Si necesitáramos hacer uso de ella varias veces deberíamos haberla almacenado en una variable.

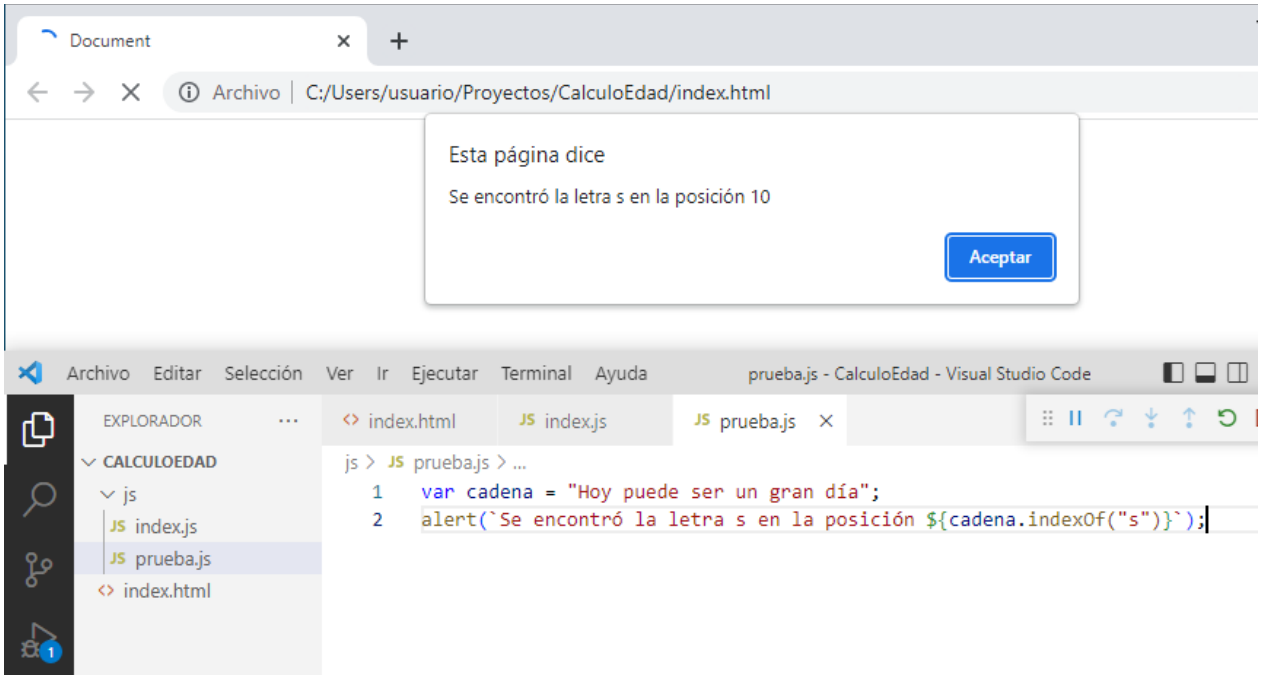
Vamos a modificar el código para no tener que pedir por separado el año, el mes y el día. Vamos a pedir al usuario que introduzca su fecha de nacimiento en formato **dd/mm/aaaa**. Vamos a utilizar métodos de la clase **String** para obtener de la cadena los dígitos correspondientes al día, mes y año y ver si son correctos.

La clase **String** tiene multitud de métodos para trabajar con cadenas. En **https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String** encontraremos la referencia completa de la clase.

Se supone que vamos a tener una cadena en donde el día puede tener un dígito o dos, lo mismo que el mes, y que el año va a tener 4 cifras. La clase **String** tiene un método llamado **indexOf** que permite buscar una subcadena dentro de una cadena y devolviendo la posición de inicio de la subcadena en la cadena. Los caracteres dentro de una cadena están referenciados por un índice que comienza en 0 y llega hasta la longitud **-1.** Así en la cadena **"Salamanca"** el carácter con índice **0** es el carácter **S**. Se puede acceder a un carácter en cualquier cadena sabiendo su índice poniendo tras el nombre de la variable o de la cadena un par de corchetes **[ ]** y en su interior el número del índice. Por ejemplo para referirnos al carácter **m** de la cadena **"Salamanca"** podemos poner **"Salamanca"[4]**. Esta terminología de utilizar corchetes para referirnos a un elemento entre muchos es la misma que se utiliza para acceder a los elementos de algo, que todavía no hemos visto, que son los **array**, tabla o vector. Podemos decir que una cadena se comporta como un array de caracteres.

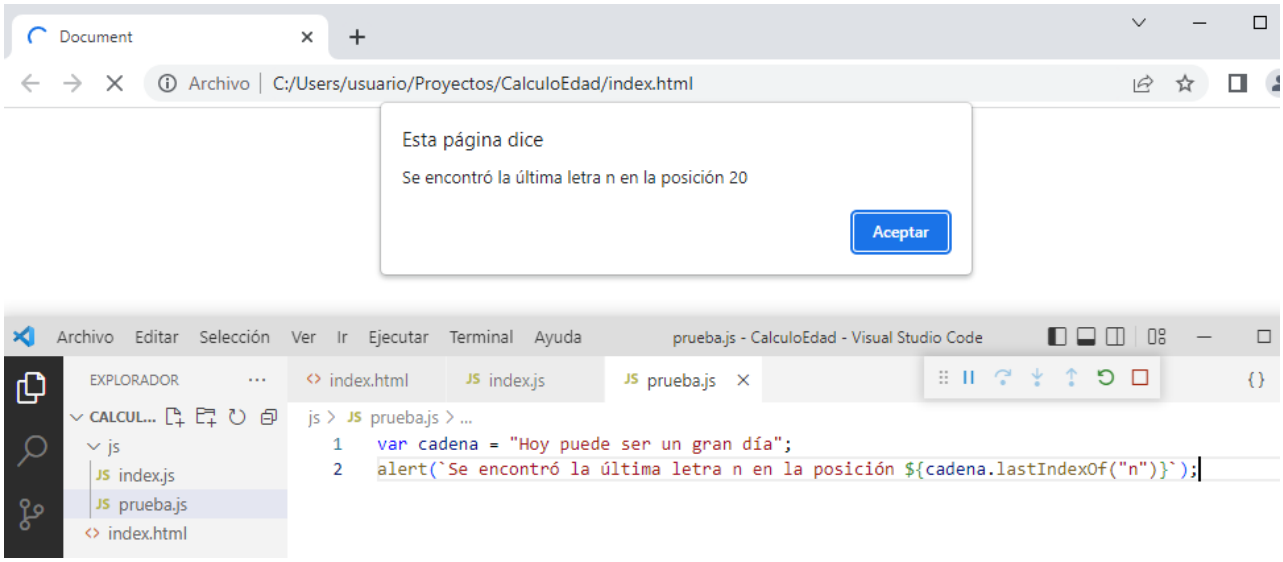
El siguiente es un ejemplo del uso de **indexOf:**

```
var cadena = "Hoy puede ser un gran día";
alert(`Se encontró la letra s en la posición ${cadena.indexOf("s")}`);
```



La clase **String** tiene otro método que se llama **lastIndexOf** que hace lo mismo pero desde el final de la cadena hacia el comienzo:

```
var cadena = "Hoy puede ser un gran día";
alert(`Se encontró la última letra n en la posición ${cadena.lastIndexOf("n")}`);
```



Y por último vamos a necesitar un método que nos permita extraer una subcadena de una cadena dando su posición de inicio y la de fin; este método es **substring**.

Las cadenas tienen una propiedad **length** que nos devuelve el número de caracteres que la forman.

El proceso será: localizamos el primer slash, los caracteres que hay antes de él en la cadena son los caracteres correspondientes al día. Localizamos la posición del último slash, los caracteres que hay desde él hasta el final son los correspondientes al año. Por último, los caracteres que hay entre los dos slash serán los correspondientes al mes. Si el usuario no escribiera dos slash exactamente el fallo será detectado cuando comprobemos los componentes día, mes y año.

```
var edad=0;
var fechaNacimiento="";
var añoNacimiento=0;
var mesNacimiento=0;
var díaNacimiento=0;
var fechaActual = new Date();
var añoActual = fechaActual.getFullYear();
var mesActual = fechaActual.getMonth() + 1;  // porque en javascript el mes va de 0 a 11
var díaActual = fechaActual.getDate();
var indicePrimerSlash, indiceSegundoSlash;
fechaNacimiento = prompt("¿cual es tu fecha de nacimiento?(dd/mm/aaaa)");
if (fechaNacimiento != null) {   // no se pulsó cancelar
   indicePrimerSlash = fechaNacimiento.indexOf("/");
   indiceSegundoSlash = fechaNacimiento.lastIndexOf("/");
   díaNacimiento= parseInt(fechaNacimiento.substring(0,indicePrimerSlash));
   mesNacimiento = parseInt(fechaNacimiento.substring(indicePrimerSlash+1,
                                                              indiceSegundoSlash)) ;
   añoNacimiento =parseInt(fechaNacimiento.substring(indiceSegundoSlash+1));
   if (isNaN(díaNacimiento) || isNaN(mesNacimiento) ||
                                 isNaN(añoNacimiento))
      alert("el formato de fecha introducido es incorrecto: dd/mm/aaaa");
   else
      if (añoNacimiento < 1900 ||
         añoNacimiento > añoActual)
         alert("para hacer el cálculo es necesario introducir un año entre 1900 y el año actual");
      else {
         if( mesNacimiento < 1 ||
               mesNacimiento > 12)
               alert("para hacer el cálculo es necesario introducir un mes entre 1 y el 12");
            else {
               if (  díaNacimiento < 1 ||
                        (díaNacimiento+"/"+mesNacimiento+"/"+añoNacimiento)
                            != (new Date(añoNacimiento,mesNacimiento -1,
                                                      díaNacimiento)).toLocaleDateString("es-ES"))
                  {
                     let díasMes= (new Date(añoNacimiento,mesNacimiento,0)).getDate();
                      alert(`el día para el mes introducido no está entre 1 y ${díasMes}`);
                  }
                else {
                     // año mes y día están entre límites
                     edad = añoActual - añoNacimiento;
                     // si mes y día de la fecha de nacimiento es mayor que mes y día de la
                     // fecha actual hay que corregir en 1                                    if i
                     if(mesNacimiento > mesActual ||
                              (mesNacimiento == mesActual
                                 && díaNacimiento > díaActual))
```

```
                    edad--;
                alert(`tienes ${edad} años de edad`);
                if (mesNacimiento == mesActual &&
                        díaNacimiento == díaActual)
                    alert("Felíz cumpleaños");
            }

        }
    }
}
```

El código se ha simplificado algo, pero sigue habiendo alguna laguna. Por ejemplo, no detectaría como incorrecta la introducción de **00002/000004/0002020**, tampoco **20/12.5/2020.56**.

Como se lleva viendo en los distintos ejercicios que vamos viendo se gasta mucho código en **validar** lo que un usuario introduce en la petición de un dato. Si nos fijamos, el cálculo de la edad, si el usuario introdujese bien el dato, sería muy pocas instrucciones. El **filtrado de datos** es algo que hay que hacer de forma **obligatoria** para impedir que una mala entrada del usuario rompa nuestro código dando la posibilidad de que un usuario con mala fe se aproveche de los mensajes que emite el intérprete cuando detecta un error de ejecución para acceder por ejemplo a información confidencial, o acceso indebido al sistema o a información contenida en bases de datos. Por suerte para el programador, existe una herramienta muy eficiente que nos ayuda a realizar este filtrado de una forma más sencilla, son las llamadas **expresiones regulares**.

Expresiones regulares

Una expresión regular no es más que un **patrón de caracteres** con significado especial que aplicado a una cadena de texto nos indica la concordancia o no con él. Por ejemplo, podemos establecer un patrón que indique la concordancia con solo números. Si aplicamos este patrón a una cadena podemos saber si la cadena está compuesta sólo por números. Otro ejemplo, podemos establecer un patrón para que indique la concordancia con una cadena de entre 6 y 10 caracteres que pueden ser letras mayúsculas y guiones solamente. Las posibilidades son enormes. Un último ejemplo, podemos establecer un patrón para que la concordancia sea con una cadena de entre 8 y 15 caracteres de entre letras, mayúscula o minúsculas, dígitos y caracteres especiales y que además haya un elemento de cada grupo. Pueden ser los requisitos de una contraseña. Una vez escrito el patrón la concordancia se resuelve con una simple instrucción.

En javaScript una expresión regular, como casi todo tipo de dato, tiene categoría de objeto. Tenemos una clase que se llama **RegExp** que encapsula toda la funcionalidad. El constructor recibe como argumento el patrón de la misma. Al igual que con las cadenas podemos escribir una expresión regular sin tener que instanciar de la clase escribiendo el patrón entre caracteres **slash / /.**

En el patrón se escriben caracteres y algunos de ellos tienen un significado especial. Las letras mayúsculas y minúsculas no tienen un significado especial. Si dentro de un patrón aparece alguna de ellas significa concordancia con la letra. Por ejemplo, el patrón "a" tiene concordancia con cualquier letra **a** en la cadena a la que se aplica la expresión regular:

```
var expresión = new RegExp("a");
```

```
console.log(expresión.test("Salamanca"));    // aparecerá true
console.log(expresión.test("Teruel"));       // aparecerá false
```

También podría haber sido escrito:

```
var expresión = /a/;
console.log(expresión.test("Salamanca"));    // aparecerá true
console.log(expresión.test("Teruel"));       // aparecerá false
```

o:

```
console.log(/a/.test("Salamanca"));                 // aparecerá true
console.log((new RegExp("a")).test("Teruel"));      // aparecerá false
```

Si la cadena contuviera varias letras, las concordancia sería con todas ellas y en la secuencia en que aparecen:

```
var expresión = /ala/;
console.log(expresión.test("Salamanca"));    // aparecerá true
console.log(expresión.test("camelia"));      // aparecerá false
```

La concordancia es exacta con el carácter, por lo tanto se distinguen mayúsculas de minúsculas. Si no se desea distinguir se debe añadir un **flag**, una marca, en la expresión regular, la letra **i**.

```
console.log(/s/.test("Salamanca"));     // aparecerá false
console.log(/s/i.test("Salamanca"));    // aparecerá true
```

Si se hubiera instanciado de la clase, los flag se ponen como segundo argumento, tras el patrón.

```
var expresión = new RegExp("a","i")
console.log(expresión.test("Salamanca"));   // aparecerá true
```

Los dígitos en principio no tienen significado especial a no ser que aparezcan entre llaves, que si que tienen significado especial. Por ejemplo el siguiente comprobaría si en la cadena está la matrícula **1234-FJS**.

```
console.log(/1234-FJS/.test("Mi primer coche tenía de matrícula 1234-FJS y
      el segundo 4321-SJF"));   // aparecerá true
```

Si en una determinada posición de la cadena del patrón quisiéramos la **concordancia con uno** de entre varios caracteres los enumeramos encerrándolos entre corchetes, que tienen un significado especial en las expresiones regulares. Por ejemplo, el siguiente coincidiría con **camellos** y con **camellas**:

```
console.log(/camell[ao]s/.test("Cuento 12 camellos en fila"));   // aparecerá true
```

Si los caracteres incluidos entre corchetes son consecutivos en el orden Unicode se puede poner el **primero, un guión y el último** ahorrándonos de escribirlos todos. Por ejemplo en el siguiente habrá concordancia con matrículas que comiencen por **1234-FJ** y a continuación una letra mayúscula entre la **P** y la **T**.

```
console.log(/1234-FJ[P-T]/.test("Mi primer coche tenía de matrícula 1234-FJS"));   // aparecerá true
```

Se pueden incluir varios rangos de caracteres. En el siguiente ejemplo la concordancia será con cualquier letra mayúscula del alfabeto ingles y cualquier dígito:

```
console.log(/código:[0-9A-Z]xxx[0-9]/.test("la contraseña es código:Bxxx7, y es secreta"));
                        // aparecerá true
```

Si la concordancia fuera con un carácter que tiene significado especial dentro de la expresión regular, por ejemplo un corchete o el guión, se pondrá escapado, que ya sabemos por las cadenas y las comillas, que es escribirlo precedido de un backslash ****.

```
console.log(/código:[0-9A-Z\[]xxx[0-9]/.test("la contraseña es código:[xxx7, y es
                        secreta"));   // aparecerá true
```

Cuando uno de los elementos del patrón aparece repetido, en lugar de escribirlo múltiples veces podemos escribir este elemento seguido de un par de llaves y el número de veces que está repetido. Por ejemplo el siguiente podría ser un patrón para que concuerde con un nif de 8 dígitos y una letra mayúscula del alfabeto inglés.

```
console.log(/[0-9]{8}[A-Z]/.test("mi nif es 12345678H"));   // aparecerá true
```

Se puede indicar el número mínimo y máximo de repeticiones poniendo ambos valores separados por coma entre los paréntesis. En el ejemplo siguiente se permite poner 7 u 8 dígitos en el nif:

```
console.log(/[0-9]{7,8}[A-Z]/.test("mi nif es 1234567H"));   // aparecerá true
```

Las llaves de repetición solo afectan al elemento más cercano que lo precede. Si queremos que afecte varios elementos encerraremos estos entre paréntesis **( )**. El siguiente ejemplo muestra concordancia con una dirección MAC:

```
console.log(/[0-9A-F]{2}(-[0-9A-F]{2}){5}/.test("dirección MAC A5-00-00-45-FF-69 del
                        dispositivo"));   // aparecerá true
```

Nótese que no se ha escapado el carácter guion porque no está dentro de los corchetes en donde si tendría un significado especial.

El carácter punto tiene un significado especial que es el de concordancia con cualquier carácter. Por ejemplo el siguiente patrón tendrá concordancia con cualquier subcadena de cinco caracteres de longitud que comience con una **B** o una **V** y termine en una vocal:

```
console.log(/[BV]...[aeiou]/.test("Nació en Valladolid en 1929"));   // aparecerá true
```

También podríamos haber escrito el patrón así: **/[BV]**.{3}**[aeiou]/**

Los caracteres **? +** y ***** tienen también significado especial:

- **?** Indica que el elemento anterior puede estar repetido **0** u **1** vez, sería equivalente a **{0,1}**.

- **+** Indica que el elemento anterior puede estar repetido **0** o **más veces**, sería equivalente a **{0,}**.
- **+** Indica que el elemento anterior puede estar repetido **1 o más veces**, sería equivalente a **{1,}**.

Los patrones que hemos visto hasta ahora tienen concordancia con una subcadena en cualquier posición de la cadena a la que se le aplica. Podemos indicar la concordancia al inicio de la cadena poniendo como primer carácter del patrón el carácter acento circunflejo **^**.

```
console.log(/^Salamanca/.test("Plaza Mayor de Salamanca"));   // aparecerá false
console.log(/^Salamanca/.test("Salamanca Ciudad Europea de la Cultura"));
                                    // aparecerá true
```

También podemos ver la concordancia de la subcadena al final de la cadena incluyendo como último carácter del patrón el carácter dólar **$.**

```
console.log(/Salamanca$/.test("Plaza Mayor de Salamanca"));   // aparecerá true
console.log(/Salamanca$/.test("Salamanca Ciudad Europea de la Cultura"));
                                    // aparecerá false
```

Si se incluyen los dos la concordancia será de igualdad entre la subcadena del patrón y la cadena:

```
console.log(/^[0-9]{7,8}[A-Z]$/.test("mi nif es 1234567H"));    // aparecerá false
console.log(/^[0-9]{7,8}[A-Z]$/.test("1234567H"));              // aparecerá true
console.log(/[0-9]{7,8}[A-Z]/.test("mi nif es 1234567H"));      // aparecerá true
```

Se puede también buscar concordancia en límite de palabra haciendo uso del par de caracteres **\b**

```
console.log(/por/.test("Le dio un porrazo"));                  // aparecerá true
console.log(/\bpor/.test("Le dio un porrazo"));                // aparecerá true
console.log(/\bpor\b/.test("Le dio un porrazo"));              // aparecerá false
console.log(/por\b/.test("Esto es importante"));               // aparecerá false
console.log(/\bpor\b/.test("Es por esto que te quiero"));      // aparecerá true
```

En las expresiones regulares se utiliza el carácter pipe **|** para permitir la concordancia con una de dos subcadenas. Por ejemplo: **http|ftp** indica concordancia bien con la subcadena **http** bien con la subcadena **ftp.** Se debe tener cuidado porque este carácter toma todo lo que está a su izquierda y todo lo que está a su derecha como subcadenas para la concordancia. Con el ejemplo se ve:

```
console.log(/^http|ftp/.test("http es protocolo para hipertexto"));    // aparecerá true
console.log(/^http|ftp/.test("El protocolo para hipertexto http"));    // aparecerá false
console.log(/^http|ftp/.test("El protocolo para trasferir archivos es ftp"));
                                    // aparecerá true
console.log(/^(http|ftp)/.test("El protocolo para trasferir archivos es ftp"));
                                    // aparecerá false
```

El anterior patrón se interpreta como que comienza por la subcadena **http** o contiene la subcadena **ftp**. La forma correcta para pedir concordancia sobre una cadena que comienza por **http** o comienza por **ftp** es la última. Nótese el uso de los paréntesis para restringir el ámbito del carácter **|**

¿Cual sería el patrón para filtrar una fecha en formato **dd/mm/aaaa** en el que el día puede estar puesto con una o dos cifras comprendidas entre **1** y **31**, el mes entre **1** y **12** y el año entre **1900** y **2099:**

- Para el día entre **1 y 31** con una o dos cifras podría ser: **[0-3]?[0-9],** pero nos permitiría el **00** y el **39** por ejemplo. Habrá que dividir los rangos. Si es de una cifra o empieza por **0** nos valdría: **0?[1-9]** . Para las fecha de dos cifras que no comiencen por **3** nos valdría: **[12][0-9]**. Por último nos quedarían las que empiezan por **3: 3[01]**. Juntándolo todo tenemos:

  ```
  0?[1-9]|[12][0-9]|3[01]
  ```

  para filtrar el día.

- Con el mes pasa algo parecido. Para los meses de una cifra con un **0** opcional tendríamos: **0?[1-9] .** Y para los meses de dos cifras: **1[0-2].** Juntándolo:

  ```
  0?[1-9]|1[0-2]
  ```

- Por último el año. Siempre serán cuatro cifras en donde las dos primera pueden ser o **19** o **20** y las dos últimas pueden variar entre **00** y **99:**

  ```
  (19|20)[0-9]{2}
  ```

Juntándolo todo nos quedará como expresión regular:

```
/^(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[0-2])\/((19|20)[0-9]{2})$/
```

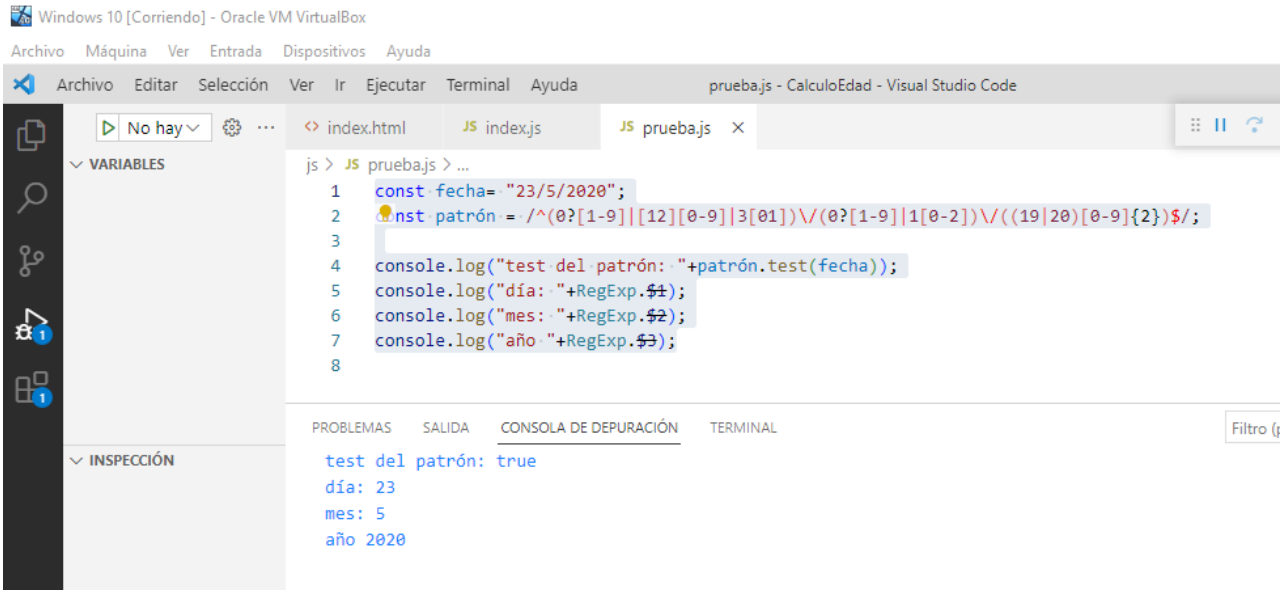
Nótese como se han utilizado los paréntesis para agrupar y restringir el ámbito del carácter de **pipe |** y como se ha escapado el carácter **slash /** para impedir que se cierre la expresión regular en la separación de día mes y anio.

Esta expresión regular nos ahorrará mucho código cuando la incorporemos a nuestro programa pero no nos libera de ver si los datos suministrados se corresponde con una fecha mal compuesta como por ejemplo un **31 de abril** o un **29 de febrero** de un año no bisiesto. Necesitamos seguir extrayendo el día, el mes y el año para realizar los cálculos posteriores. Parece que el ahorro entonces no va a ser tanto, pero no es así. En las expresiones regulares hay acceso a las concordancias a través de lo que se llaman **subexpresiones**. Cada vez que se utilizan los paréntesis para agrupar elementos se crea una subexpresión. En la última expresión regular que hemos escrito el primer grupo de paréntesis crea la subexpresión 1, el segundo la 2 etc.

En javaScript una vez hecho el **test** tenemos acceso a la concordancia de las subexpresiones a través de las propiedades **estáticas $**x de la clase **RegExp.** Una propiedad de una clase es estática cuando pertenece a la clase, no a las instancias como son todas las que hemos visto hasta ahora. Una instancia de **RegExp** no tiene propiedad **$1** por ejemplo y la tiene la clase **RegExp**. En el siguiente ejemplo se comprueba que la fecha **23/5/2020** se ajusta al patrón y se muestran sus componentes de día mes y año.

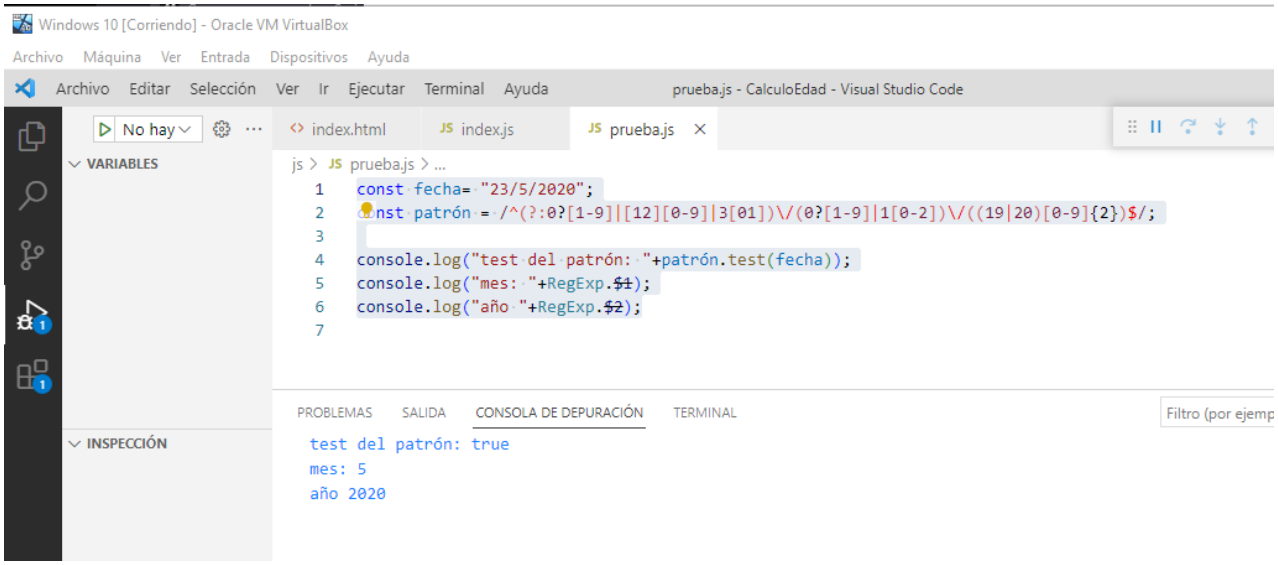
```
const fecha= "23/5/2020";
const patrón = /^(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[0-2])\/((19|20)[0-9]{2})$/;
```

```
console.log("test del patrón: "+patrón.test(fecha));
console.log("día: "+RegExp.$1);
console.log("mes: "+RegExp.$2);
console.log("año "+RegExp.$3);
```



Hay ocasiones en las que se desea que un grupo de paréntesis no se convierta en una expresión y que pueble una de las variables estáticas de la clase **RegExp**. Por ejemplo, solo queremos extraer el mes y el año de la fecha en el ejemplo anterior. No necesitamos que en **$1** esté el día. Para esto no hay más que poner **?:** tras el paréntesis de apertura del grupo. El siguiente ejemplo muestra su uso:

```
const fecha= "23/5/2020";
const patrón = /^(?:0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[0-2])\/((19|20)[0-9]{2})$/;

console.log("test del patrón: "+patrón.test(fecha));
console.log("mes: "+RegExp.$1);
console.log("año "+RegExp.$2);
```



Tenemos una referencia completa y ejemplos de uso en
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions

NOTA: Las referencias a las concordancias de las subexpresiones en una expresión regular a través de las propiedades estáticas se han marcado como **deprecated** y se debe evitar su uso. En su lugar se deben usar los métodos **match** o **matchAll** que devuelven un array con las coincidencias. Como aún no se ha hablado de la definición y uso de los array se ha preferido en este momento el uso de las propiedades estáticas para la explicación de las subexpresiones y captura de las coincidencias

Ya sabemos lo suficiente para rehacer nuestro código haciendo uso de lo aprendido.

```
var edad=0;
var fechaNacimiento="";
var añoNacimiento=0;
var mesNacimiento=0;
var díaNacimiento=0;
const fechaActual = new Date();
const añoActual = fechaActual.getFullYear();
const mesActual = fechaActual.getMonth() + 1;  // porque en javascript el mes va de 0 a 11
const díaActual = fechaActual.getDate();
fechaNacimiento = prompt("¿cual es tu fecha de nacimiento?(dd/mm/aaaa)");
if (fechaNacimiento != null) {   // se pulsó cancelar
      if (! /^(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[0-2])\/((19|20)[0-9]{2})$/
                                        .test(fechaNacimiento))
        alert("el formato de fecha introducido es incorrecto: dd/mm/aaaa");
    else {
        díaNacimiento = parseInt(RegExp.$1);
        mesNacimiento =  parseInt(RegExp.$2);
        añoNacimiento =  parseInt(RegExp.$3);
        if ((díaNacimiento+"/"+mesNacimiento+"/"+añoNacimiento) !=
              (new Date(añoNacimiento,mesNacimiento -1, díaNacimiento)).toLocaleDateString("es-ES"))
           {
              let díasMes= (new Date(añoNacimiento,mesNacimiento,0)).getDate();
              alert(`el día para el mes introducido no está entre 1 y ${díasMes}`);
           }
        else {
```

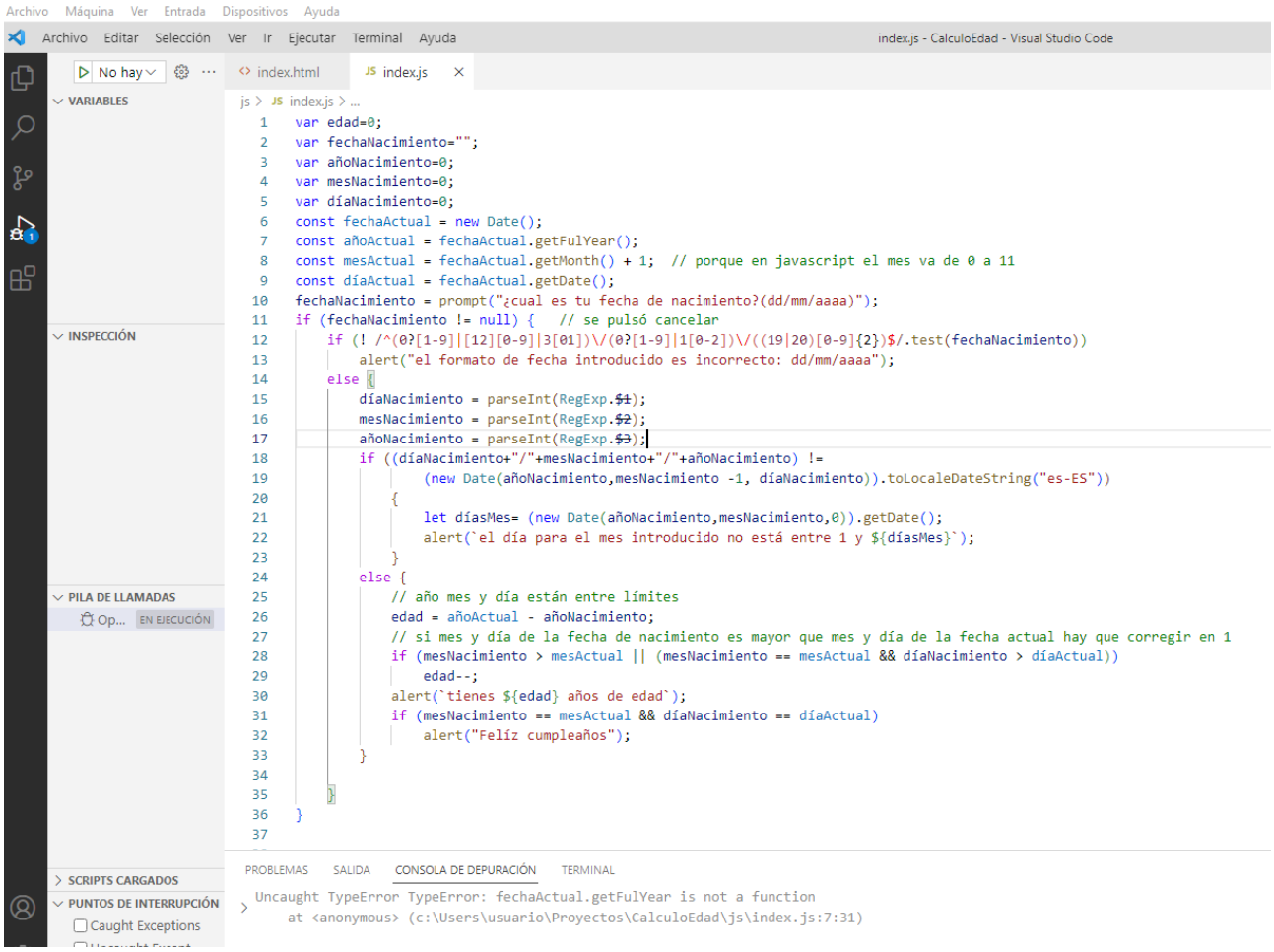
```
            // año mes y día están entre límites
            edad = añoActual - añoNacimiento;
              // si mes y día de la fecha de nacimiento es mayor que mes y día de la fecha actual
              //hay que corregir en 1
            if (mesNacimiento > mesActual ||
                  (mesNacimiento == mesActual && díaNacimiento > díaActual))
               edad--;
            alert(`tienes ${edad} años de edad`);
            if (mesNacimiento == mesActual && díaNacimiento == díaActual)
               alert("Felíz cumpleaños");
         }

      }
   }
```

Depuración. Debug

Cuando escribimos código para resolver un problema podemos cometer dos tipos de errores: los **sintácticos** y los **lógicos**.

La detección y arreglo de los errores sintácticos se hace con la ayuda del intérprete que es el que comprueba la corrección de la sintaxis y si detecta algún fallo nos mostrará un mensaje de error en donde se verá además la línea en que se produjo el error deteniendo la ejecución del programa.

En la siguiente captura se muestra el error por escribir en la línea 7 del código anterior **getFulYear** en lugar de **getFullYear**

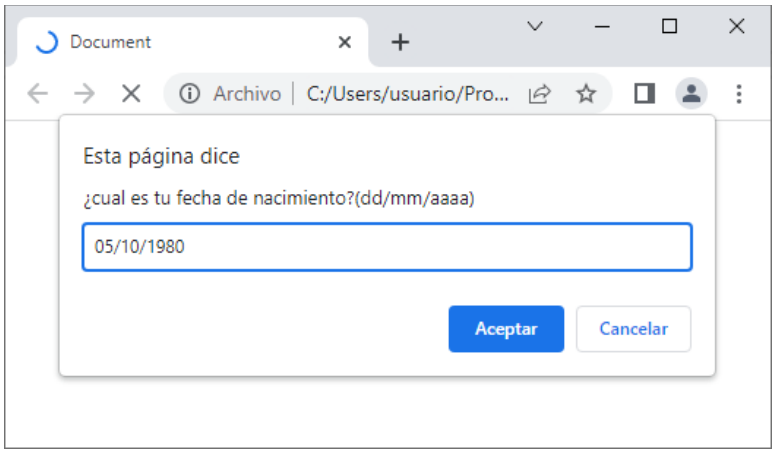


Nos indica que en la línea 7 carácter 31 se ha detectado un error **fechaActual.getFulYear is not a function**.

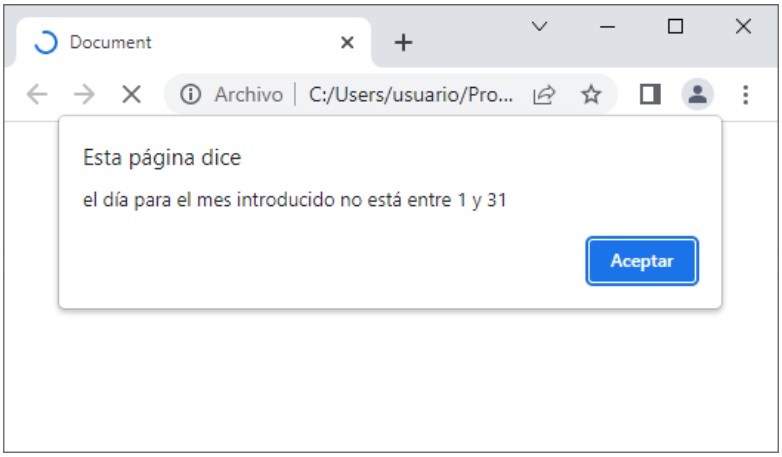
La corrección se realizar sobre el editor de texto ayudándonos de documentación técnica de sintaxis, por ejemplo del MDN de Mozilla en

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

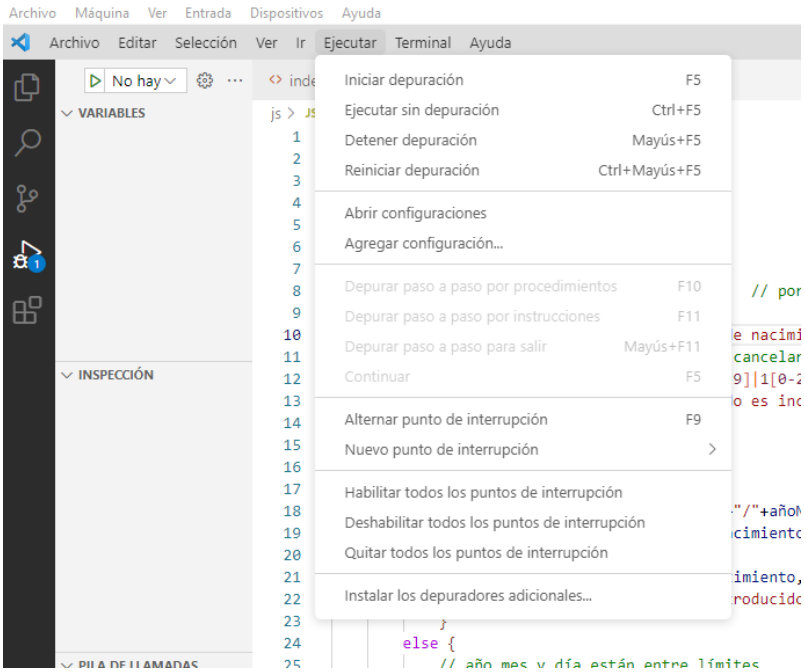
La detección y corrección de los errores lógicos es un poco más complicada, porque la sintaxis es correcta y lo que pasa es que el algoritmo no está bien diseñado. Por ejemplo, si nuestro código no estuviera bien al introducir una fecha nos daría una edad errónea. Vamos a provocar un error lógico; en la línea 15 vamos a quitar la función **parseInt** y vamos a probar la ejecución de la fecha **05/10/1980**, y suponemos que la estamos ejecutando el **10 de octubre de 2022**. Debería indicar que se tienen **42** año ¿no?



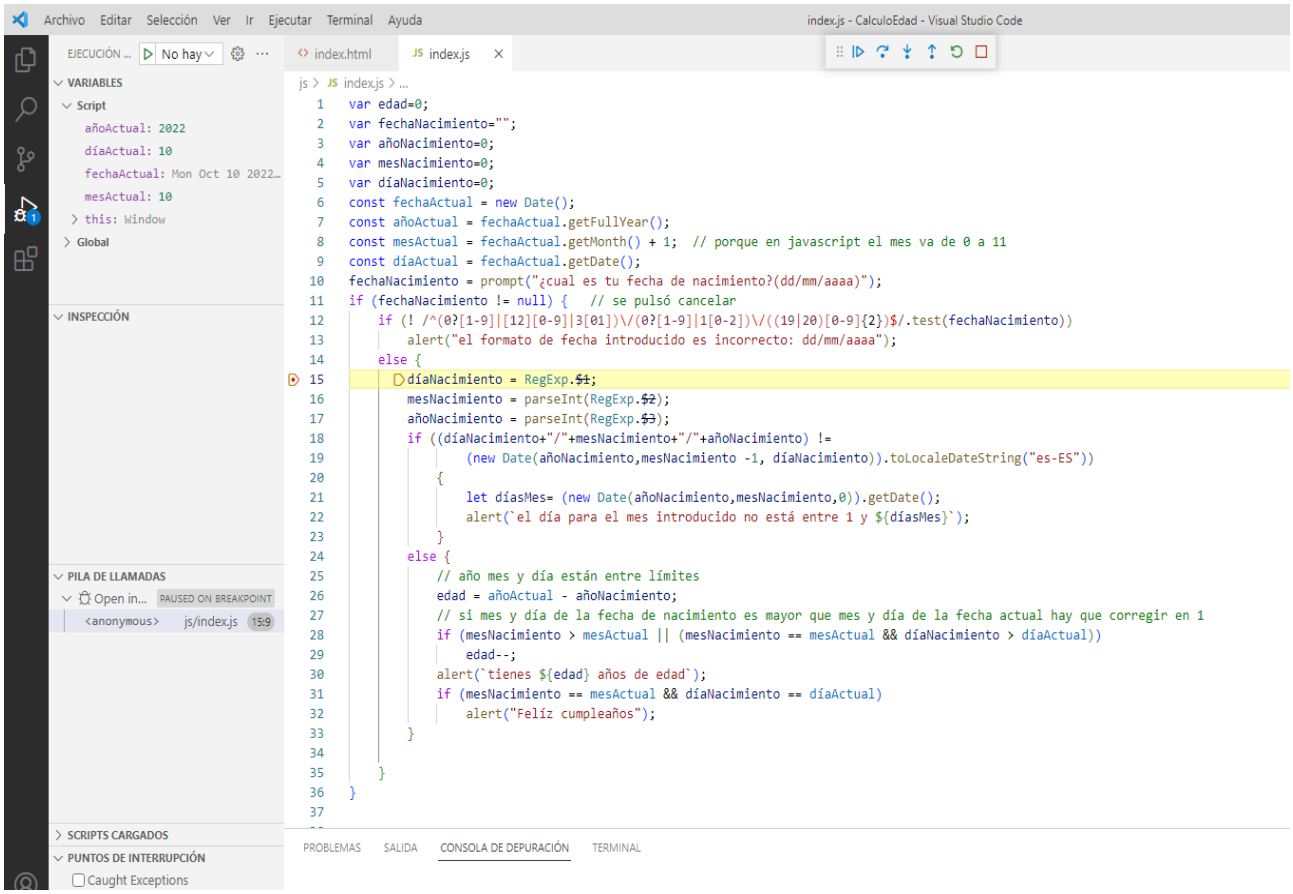
Produce una salida:



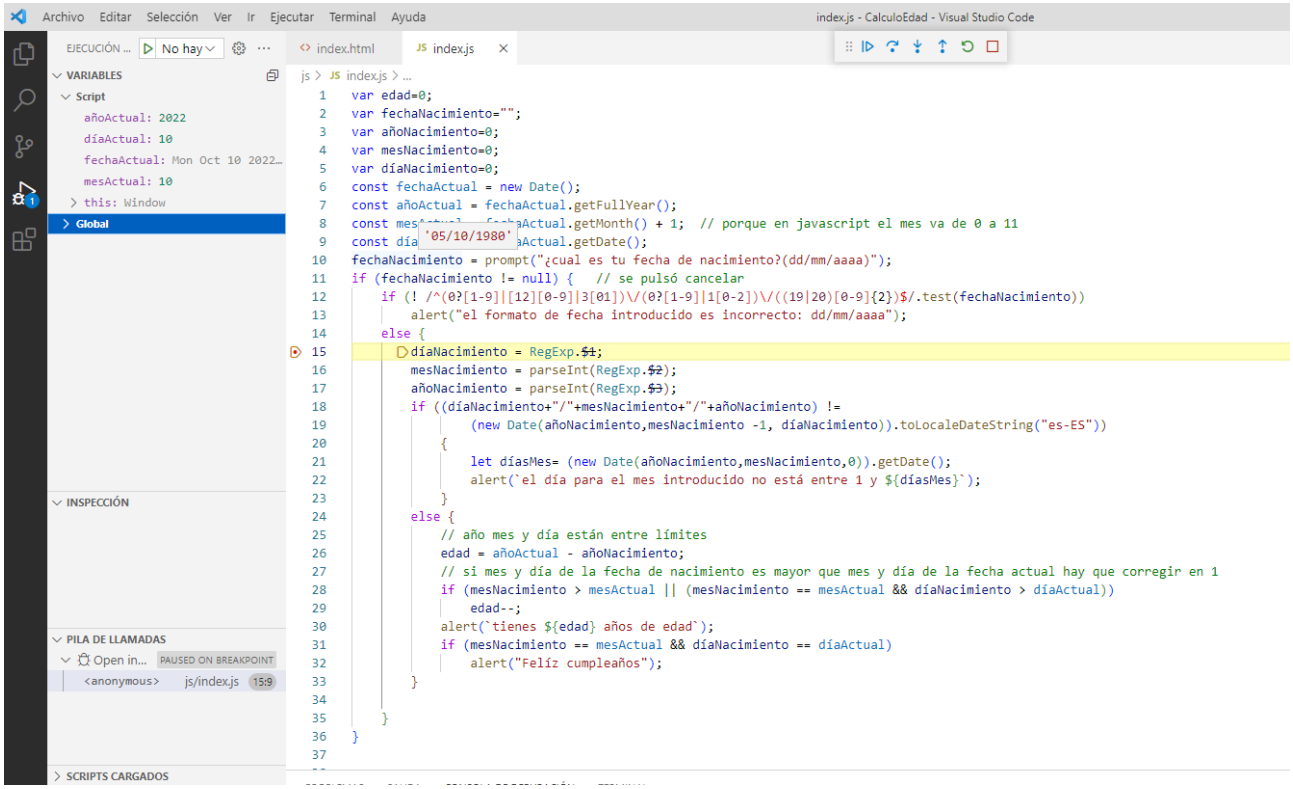
Hay algún error lógico. ¿Como localizamos el error? Visual Studio Code dispone de un depurador de código javaScript. En el menú **Ejecutar** podemos elegir la ejecución con depuración **F5** o ejecutar sin depuración **Ctrl+F5**:



Si ejecutamos de una forma u otra aparentemente conseguimos el mismo resultado. Hay una diferencia grande, si hacemos clic en el margen que queda entre la zona en gris izquierda y el número de línea de una línea de código veremos que aparece un círculo relleno en blanco. Hemos puesto un punto de parada, **breakpoint**. Si volvemos a hacer clic el punto de parada desaparecerá. Si ahora ejecutamos con **F5** veremos que cuando le llega el turno de ejecución a esa línea marcada la ejecución se detiene de forma temporal quedando destacada la línea en cuestión:



Si nos fijamos en la parte en gris de la izquierda veremos el contenido actual de las variables definidas y que son visibles en el actual ámbito y el global. Si llevamos el cursor en el código sobre alguna variable nos mostrará un tooltip con su valor actual:



En la imagen el ratón está sobre **fechaNacimiento**.

En la parte superior se muestra la caja de herramientas de la depuración con los siguientes botones:

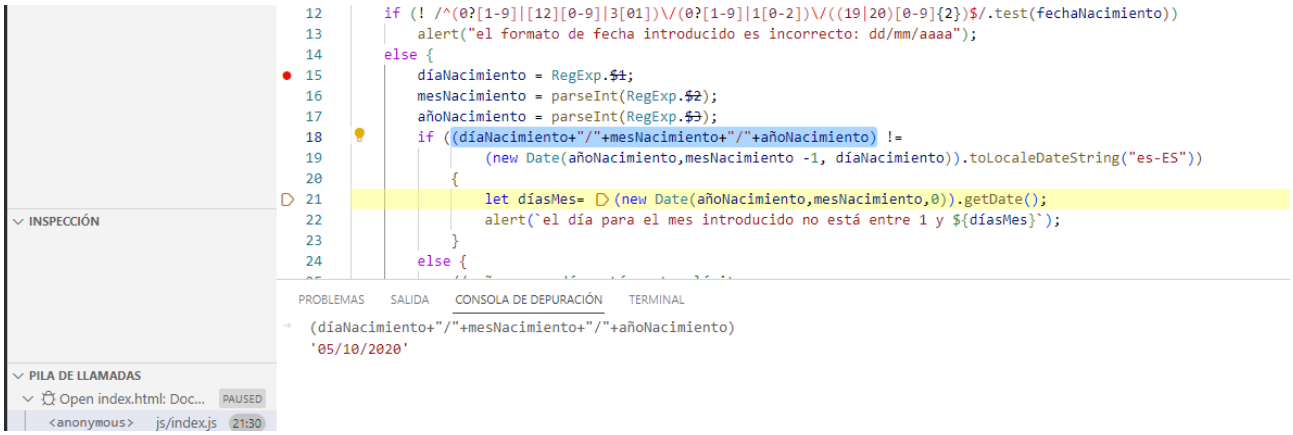
-  Ejecutar hasta el siguiente punto de interrupción.
-  Ejecutar instrucción. Si en la instrucción hubiera alguna función y método con código accesible, **no** se entrará a ejecutar su código instrucción por instrucción. Se quedará en disposición de ejecutar la siguiente instrucción.
-  Ejecutar instrucción. Si en la instrucción hubiera alguna función y método con código accesible, **si** se entrará a ejecutar su código instrucción por instrucción. Se quedará en disposición de ejecutar la siguiente instrucción de la primera función o método con código accesible.
-  Salir de depuración.
-  Reiniciar la ejecución y depuración

-  Terminar la depuración y ejecución en este momento

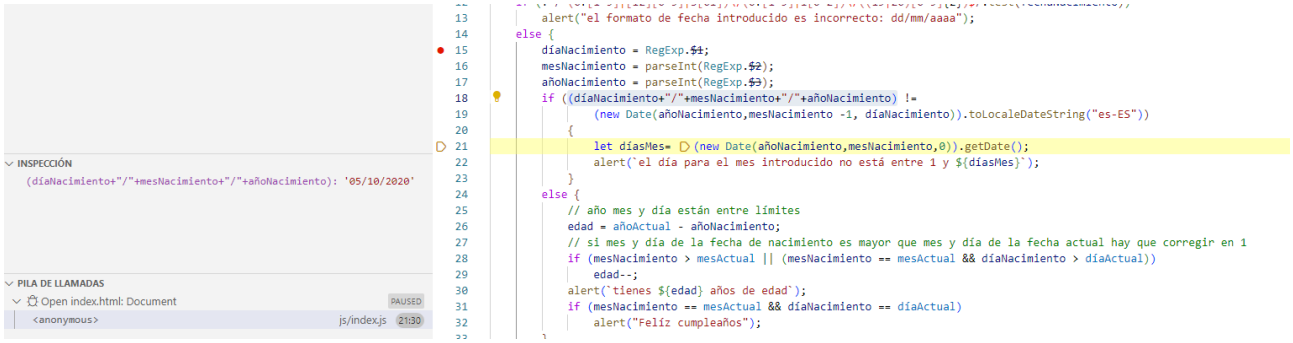
Si en el ejemplo, seguimos ejecutando paso a paso con F11 o  veremos que en el **if** ha entrado por la parte **then**. ¿Por qué ha entrado por ahí si en principio el número del día **5** es correcto para un mes **10** y un año **2020.** Analicemos la condición: por un lado compara

(díaNacimiento+"/"+mesNacimiento+"/"+añoNacimiento)

¿Como podemos ver el valor de la expresión?. La seleccionamos y pulsando con el botón derecho del ratón sobre ella seleccionamos: **Evaluar en la consola de depuración** con lo que el resultado aparecerá en dicha consola:



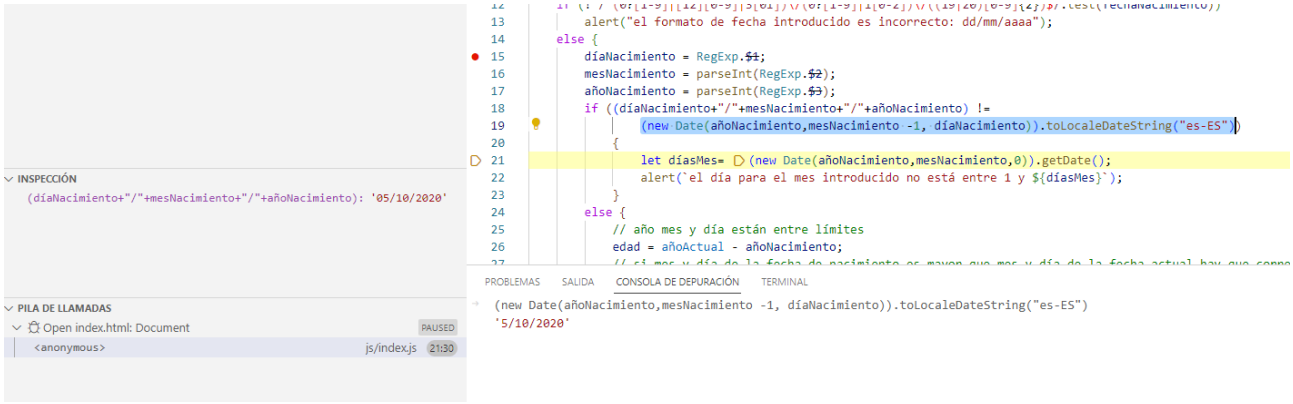
Si queremos ir viendo como va cambiando el valor de la expresión a medida que vamos ejecutando el código (no es el caso de esta expresión en donde no cambia una vez introducido el valor la primera vez) podemos insertar un **inspección,** lo mismo, pulsando con el botón derecho sobre una expresión previamente seleccionada y seleccionando **Agregar a inspección**. El resultado aparecerá en la zona de la izquierda bajo la categoría **INSPECCIÓN**.



Seguimos con la depuración. Vamos a ver el otro lado de la condición:

(new Date(añoNacimiento,mesNacimiento -1, díaNacimiento)).toLocaleDateString("es-ES")

Hacemos lo mismo, seleccionamos y seleccionamos **Evaluar en la consola de depuración** viendo:



Vemos que la primera parte de la expresión devuelve **'05/10/2020'** y la segunda **'5/10/2020'** que son claramente distintas. En la segunda parte no aparecen los ceros no significativos en el día y mes si los hubiera, mientras que en la primera pueden aparecer si el usuario los introdujo en el **prompt.** Para quitarlos la forma más fácil es convertirlos en enteros que es lo que hacía el **parseInt** que quitamos.

Las herramientas de depuración facilitan mucho la vida al programador pero es este con el análisis de su algoritmo el que debe saber que deben contener las variables y como se evalúan las condiciones en las sentencias condicionales para el correcto funcionamiento del mismo. La ¡¡¡PRÁCTICA!!! hace al maestro.

Sentencias iterativas

Nuestro programa ya hace lo que queríamos y además controla que el usuario no haga entradas incorrectas cuando introduce una fecha de nacimiento, pero tiene un defecto de funcionalidad, ¿como puede comprobar la edad de varias personas? Tal y como está volviendo a reiniciar el programa. Para permitir el cálculo de una nueva edad necesitamos volver a ejecutar casi todo el código. Es aquí donde tienen su aplicación las sentencias iterativas. Estas sentencias permiten repetir la ejecución de un bloque de código en función de una condición, mientras se siga cumpliendo la condición el bloque volverá a ser ejecutado. Tenemos tres sentencias principalmente, **while, do ... while** y **for.**

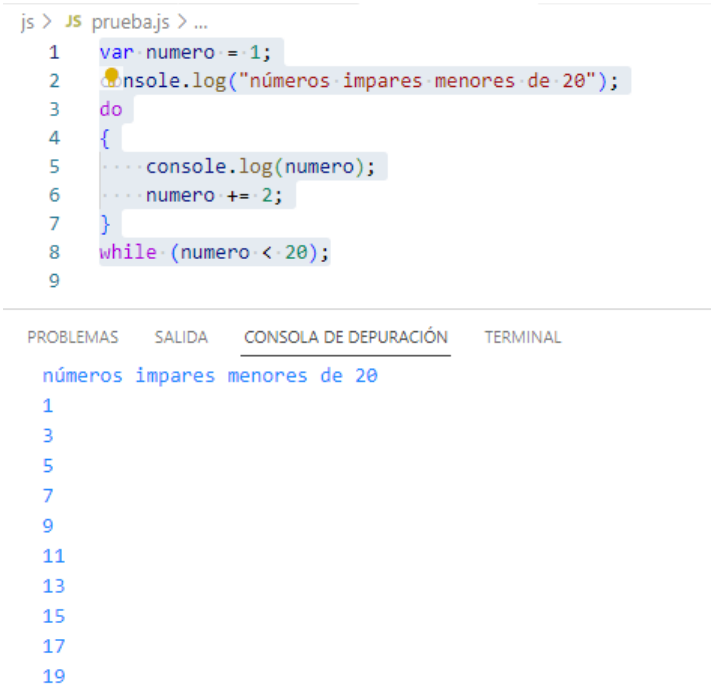
Las sentencias **while** y **do … while** son similares. La única diferencia es cuando se ve si la condición se sigue cumpliendo o no. En **while** primero se comprueba la condición y si se cumple se ejecuta el bloque de código. En **do … while** la condición se comprueba una vez ha finalizado la ejecución del bloque de código. Por lo tanto, en **do … while** el bloque de código por lo menos se ejecuta una vez.

```
var numero = 1;
console.log("números impares menores de 20");
while (numero < 20)
{
   console.log(numero);
```

```
    numero += 2;
}
```

Y con **do … while**

```
var numero = 1;
console.log("números impares menores de 20");
do
{
   console.log(numero);
   numero += 2;
}
while (numero < 20);
```



Aparentemente han funcionado igual, pero ¿y si **numero** hubiera valido, por ejemplo **25** al inicio? Se deja como ejercicio para el lector.

Vamos a modificar nuestro programa para que se soliciten nuevas fechas de nacimiento hasta que el usuario pulse en cancelar en el **prompt**.

```
var edad=0;
var fechaNacimiento="";
var añoNacimiento=0;
var mesNacimiento=0;
var díaNacimiento=0;
const fechaActual = new Date();
const añoActual = fechaActual.getFullYear();
const mesActual = fechaActual.getMonth() + 1;  // porque en javascript el mes va de 0 a 11
const díaActual = fechaActual.getDate();
var terminar = false;  // mientras no sea true se pedirán más fecha
while (! terminar)  {
   fechaNacimiento = prompt("¿cual es tu fecha de nacimiento?(dd/mm/aaaa)");
   if (fechaNacimiento == null)
      terminar=true;
   else
      {  // se pulsó cancelar
         if (! /^(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[0-2])\/((19|20)[0-9]{2})$/.test(fechaNacimiento))
            alert("el formato de fecha introducido es incorrecto: dd/mm/aaaa. \nFechas entre el 1/1/1900
y 31/12/2099");
         else {
            díaNacimiento = RegExp.$1;
            mesNacimiento = parseInt(RegExp.$2);
            añoNacimiento = parseInt(RegExp.$3);
            if ((díaNacimiento+"/"+mesNacimiento+"/"+añoNacimiento) !=
                 (new Date(añoNacimiento,mesNacimiento -1, díaNacimiento)).toLocaleDateString("es-
ES"))
               {
                  let díasMes= (new Date(añoNacimiento,mesNacimiento,0)).getDate();
                  alert(`el día para el mes introducido no está entre 1 y ${díasMes}`);
               }
            else {
               // año mes y día están entre límites
               edad = añoActual - añoNacimiento;
               // si mes y día de la fecha de nacimiento es mayor que mes y día de la fecha actual
                //hay que corregir en 1
               if (mesNacimiento > mesActual ||
                        (mesNacimiento == mesActual && díaNacimiento > díaActual))
                  edad--;
               alert(`tienes ${edad} años de edad`);
               if (mesNacimiento == mesActual && díaNacimiento == díaActual)
                  alert("Felíz cumpleaños");
            }
         }
      }
}
```

Se ha declarado una variable booleana **salir** que es puesta a **true** cuando el usuario pulsa en el botón **Cancelar** de **prompt**. Se ha modificado también el mensaje de formato de fecha erróneo para que sea más claro.

La sentencia **for** es un poco más complicada. Tras **for** y entre paréntesis se ponen tres elementos, separados por punto y coma **;**.

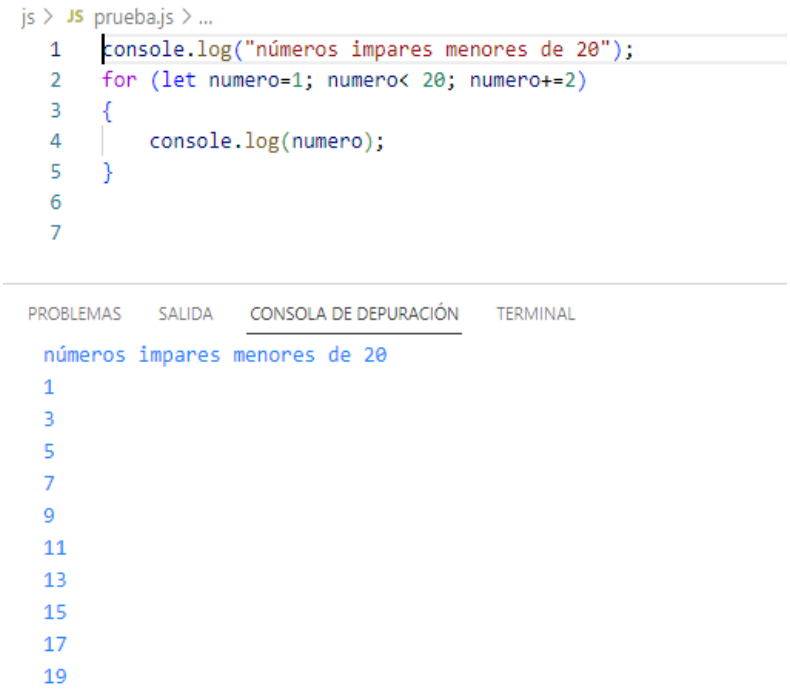
El primer elemento indica una instrucción que se ejecuta antes de ejecutar el bloque de código, viene a ser como una inicialización.

El segundo, es la condición que indica si el bloque de código se vuelve a ejecutar o no. Se evalúa antes de ejecutar el bloque por lo tanto es similar a la sentencia **while**. Mientras la condición se cumpla el bloque volverá a ser ejecutado.

El tercero es una sentencia que es ejecutada al terminar la ejecución del bloque de código que normalmente modificará la condición de termino de las iteraciones.

El siguiente ejemplo repite el ejemplo del **while**.

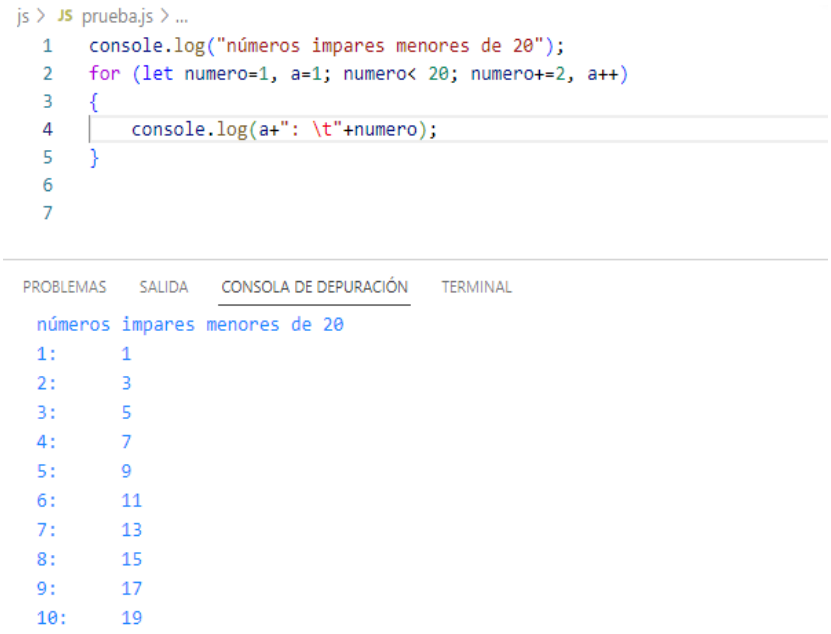
```
console.log("números impares menores de 20");
for (let numero=1; numero< 20; numero+=2)
{
   console.log(numero);
}
```



El proceso sería el siguiente: Al inicio se declara una variable **numero,** solo visible dentro del bloque de código del **for,** y se inicializa a **1**. Se comprueba que cumple la condición y como **1** es menor de **20** , se ejecutará el bloque de código, que escribirá en la consola el número **1.** A continuación se ejecutará la sentencia de fin de bloque **numero+=2** y se vuelve al comienzo, evaluando de nuevo la condición con valor de **numero= 3**, y se vuelve a ejecutar el bloque de código. Así hasta que **numero** valga **19** en que, tras mostrarse en consola, se incrementará en **2,** pasando a valer **21,** al evaluar la condición ya no se cumple y se da por finalizado el bucle.

En cada uno de los tres elementos de **for** pueden aparecer varias sentencias separadas por coma **,**

```
console.log("números impares menores de 20");
for (let numero=1, a=1; numero< 20; numero+=2, a++)
{
   console.log(a+": \t"+numero);
}
```



Ya conocemos las estructuras básicas de programación: **if, while y for**. También sabemos de los tipos básicos de datos, en el caso de javaScript: **Number, String** y **Boolean** y la estructura básica de datos que es la **variable**. También tenemos un concepto muy básico de objetos sabiendo distinguir entre **clase** e **instancia** y de miembros como son las **propiedades** y los métodos. También tenemos una idea de que son las propiedades **estáticas**.

Array

Un **vector** o un **array** o **tabla** no es más que una colección de variables todas con el **mismo nombre** y encapsuladas en un único elemento. Si tenemos muchas variables con el mismo nombre, ¿cómo distinguimos una de otra? Por su número de índice. Ya hemos comentado esto al hablar de las cadenas o strings, cuando decíamos que podíamos acceder a un carácter de la cadena por su posición dentro de la cadena y que utilizábamos su número de posición, **índice,** entre corchetes. En un array es exactamente lo mismo, si tenemos un array con diez variables, la primera tendrá de índice **0** la segunda **1** y así sucesivamente. Para acceder a una de ellas no tenemos que escribir el nombre del array seguido de un par de corchetes y en su interior el número de índice.

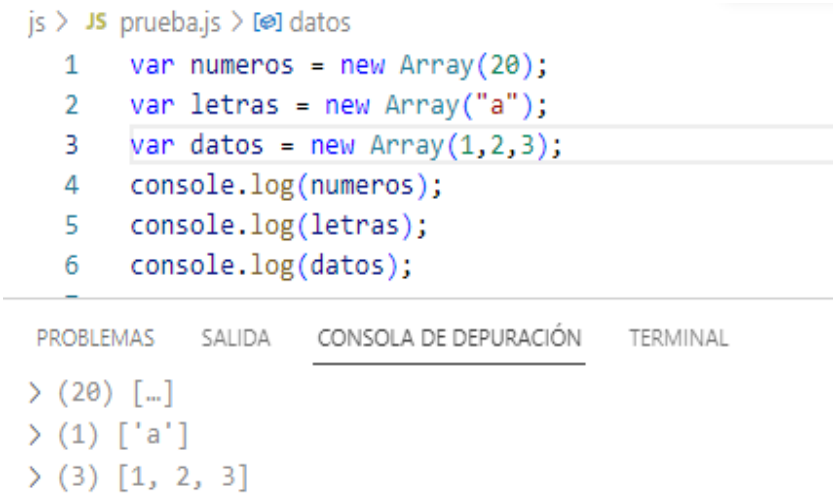
Un array se declara como cualquier otra variable con **let** o **var** seguido del nombre del array y a continuación asignándole una instancia de la clase **Array:**

```
var numeros = new Array(20);
```

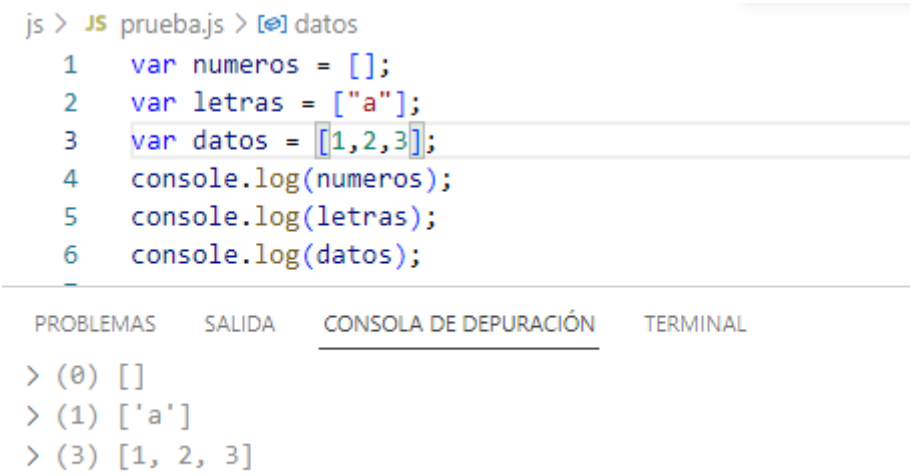
Se ha declarado un array con nombre **numeros** con **20** elementos accesibles con número de índice **0** a **19.**

Si el constructor es invocado con un argumento que no sea numérico se creará el array con un solo elemento y contenido el argumento pasado al constructor. El índice de este único elemento será **0.** Si el constructor es invocado con más de un argumento de cualquier tipo se creará el array con tantas casillas como número de argumentos y contenido el de los argumentos.

```
var numeros = new Array(20);
var letras = new Array("a");
var datos = new Array(1,2,3);
console.log(numeros);
console.log(letras);
console.log(datos);
```



Hay un forma más de declarar un array y darle contenido:



El contenido de cada casilla puede ser cualquier tipo de dato de javaScript, incluido otro array. En los lenguajes de programación fuertemente tipados todas las casillas de un array han de ser del mismo tipo de datos, en javaScript **no**.

La clase **Array** dispone de multitud de propiedades y métodos que nos facilitan su uso. En **https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array**, tenemos una referencia completa.

Html

Html es un **lenguaje de marcas** que describe la apariencia y la información que se muestra en una página web. La apariencia queda determinada, de además de la estructura descrita a través de los distintos tipos de marcas, por las llamadas hojas de estilo **CSS**. **Html no** es un lenguaje de programación y por lo tanto no es capaz de responder a las acciones del usuario más allá de modificar la apariencia de algún elemento cuando el ratón pasa sobre él o realizar una animación también de algún elemento, realizado todo esto a través de las ya mencionadas hojas de estilo. ¿cómo se consigue responder a las acciones del usuario haciendo dinámicas las páginas? Incrustando código javaScript en la misma, como ya se ha visto.

El propósito de este documento es la iniciación a la programación haciendo uso de javaScript, se deberían tener unos conocimientos básicos de diseño de páginas con **Html** y **CSS**. En la red hay muchos tutoriales básicos para tener estos conocimientos, por ejemplo en W3Schools (**https://www.w3schools.com/html)**.

Estamos pues desarrollando código javaScript en el entorno de un navegador web cuyo propósito es la visualización de páginas web. El propósito de javaScript, ya se ha dicho, en este entorno es el de dar funcionalidad a la página haciéndola dinámica, por ejemplo, respondiendo a las acciones del usuario para mostrar una información u otra. Ya hemos visto algo, el usuario escribía un fecha y el programa respondía con un mensaje indicando la edad. Tanto la entrada de información con **prompt** como la salida con **alert** se utilizan con poca frecuencia en las páginas web, la entrada se

suele hacer a través de formularios y la salida sobre la propia página. Tanto unos como los otros elementos de la página no tienen nada que ver con lo que es el lenguaje de programación. El navegador nos ofrece un entorno en el que los elementos de la página son accedidos a través de una serie de objetos, instancias más bien, que es lo que se conoce como el **DOM (D**ocument **O**bject **M**odel). Ya hemos hablado del objeto principal que es **window** a través del cual tenemos acceso a todo el contenido de la página pudiendo añadir, modificar o eliminar cualquiera de esos elementos.

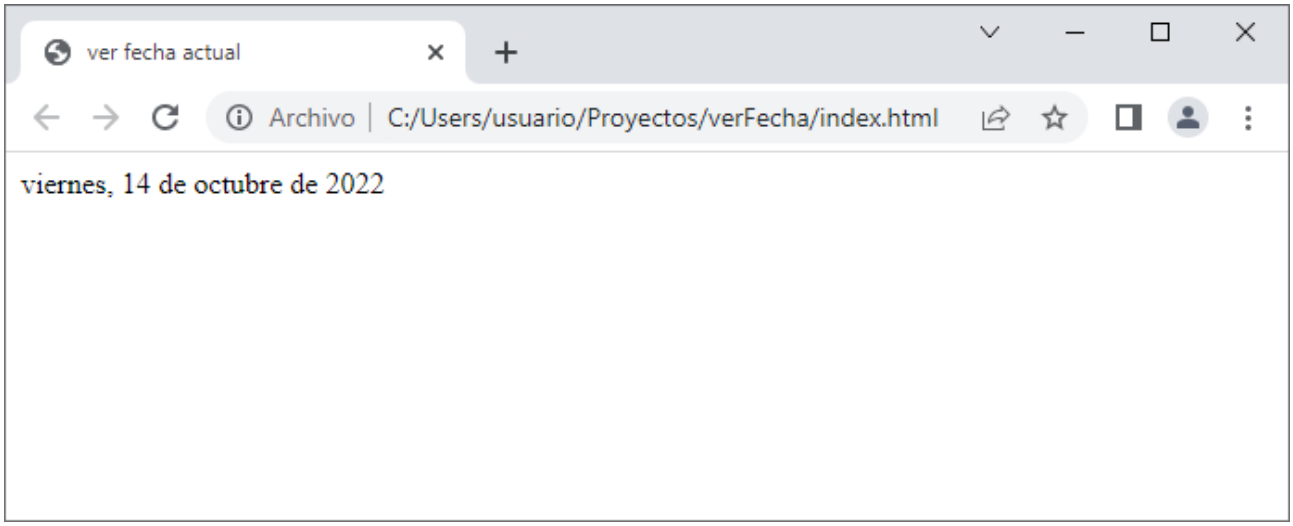
¿Como se accede a algún elemento de la página desde javaScript? Hay varias formas. La más sencilla es la de poner un atributo **ID** en el código html al elemento al que queremos acceder desde javaScript e invocar el método **getElementById** del objeto **document** que está contenido dentro del objeto **window.** En el ejemplo que sigue se ha colocado en el código html un elemento de tipo **DIV** que sabemos que es un contenedor de elementos html, le hemos puesto un atributo **ID, divFecha** y en el código javaScript hemos hecho que el contenido visible de ese elemento sea la fecha actual. Todos los elementos html que son contenedores tienen una propiedad **innerHTML, que es modificable,** con su contenido en html. En este caso el elemento **DIV** va a contener únicamente texto.

index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>ver fecha actual</title>
</head>
<body>
   <div id="divFecha"></div>
   <script src="js/index.js"></script>
</body>
</html>
```

index.js

```
var divFecha= document.getElementById("divFecha");
divFecha.innerHTML = (new Date()).toLocaleDateString("es-ES",
        {weekday:"long",day:"numeric",month:"long",year:"numeric"});
```



¿Como se sabe cuando un usuario realiza alguna acción que nos interesa para, por ejemplo, modificar el contenido de la página? Cuando un usuario realiza una acción sobre la ventana del navegador se dice que desencadena un **evento**. El navegador está en permanente escucha para detectar uno de esto eventos y cuando detecta uno mira en su lista de eventos capturados para ver si tiene que realizar alguna acción, una serie de instrucciones, que es lo que se denomina manejador, controlador o **handler** del evento. El proceso por tanto es: capturar el evento y suministrar un handler que será ejecutado cuando el navegador detecte el evento.

Funciones

Un manejador, controlador de evento o **handler** está representado por un función. Una función en cualquier lenguaje de programación, es un bloque de código representado por un nombre, de forma que el invocar o ejecutar el nombre se ejecutan las instrucciones contenidas en el bloque. Ya hemos utilizado funciones cuando hemos invocado métodos de clases predefinidas. Los métodos son funciones. Cuando hemos ejecutado **alert** en realidad hemos estado ejecutando una serie de instrucciones que han sido capaces de mostrar una nueva ventana con dos botones y que responden a pulsaciones … Las funciones pueden recibir determinados valores que serán usados en su interior, pasados a través de lo que se denominan **argumentos** que se escriben dentro de los paréntesis que siguen al nombre de la función y separados por comas.

Una función se puede definir de varias formas, una de ellas es con la palabra reservada **function** seguido del nombre de la función y la lista de parámetros que va a aceptar, separados por coma, y encerrados entre paréntesis. Una función puede devolver un valor, por ejemplo **parseInt** devuelve el valor entero del argumento pasado. Pueden no devolver nada, por ejemplo **console.log** no devuelve ningún valor, simplemente lo muestra en la consola. Para que una función devuelva un valor debe ejecutar en su interior como última sentencia una instrucción **return** seguida del valor que se desee retornar.

```
function suma(a,b)
{
  return (parseInt(a)+parseInt(b));
```

```
}
console.log(suma(4.5,20.4));  // mostrará 24
console.log(suma(23.5,10));   // mostrará 33
```

La función anterior devuelve la suma entera de dos números. En función de los parámetros suministrados en la ejecución devuelve un valor u otro.

En javaScript una función se puede considerar un tipo de dato y por lo tanto ser asignado a una variable:

```
var suma = function (a,b)
{
   return (parseInt(a)+parseInt(b));
}

var otraSuma = suma;

console.log(suma(4.5,20.4));  // mostrará 24
console.log(otraSuma(23.5,10));  // mostrará 33
```

javaScript es poco estricto en cuanto al número de argumentos que se pasan en la invocación y el número de argumentos que se pusieron en la declaración. Así **suma(23,3,12)** no provocará ningún tipo de error, simplemente el tercer argumento no es accesible dentro de la función con ningún nombre de argumento. **Suma(14)** tampoco provocará ningún tipo de error, simplemente el argumento **b** aparecerá como **undefined** dentro de la función.

Dentro de una función se tiene acceso a la variable predefinida **arguments** que es un array con los argumentos pasados en la invocación, independientemente de los argumentos que aparezcan en la declaración. Así si se invoca **suma(12,45,2), arguments** contendrá 3 casillas con valores **12, 45** y **2.** Con **suma(12)**, **arguments** contendrá una sola casilla con el valor **12**. El siguiente código expande el uso de **suma** para que pueda recibir un número indeterminado de argumentos:

```
var suma = function ()
{
   let total=0;
   for (let i=0; i< arguments.length; i++)
   {
      total+=parseInt(arguments[i]);
   }
   return (total);
}

console.log(suma());              // mostrará 0
console.log(suma(4.5));           // mostrará 4
console.log(suma(23.5,10));       // mostrará 33
console.log(suma(4.5,5,10,23)); // mostrará 42
```

De forma alternativa al uso de **arguments** en funciones con un número indeterminado de argumentos existe la sintaxis **rest** que no es más que preceder el nombre del array con un número indeterminado de argumentos, por tres puntos. Solo puede aparecer un argumento con sintaxis **rest** y debe ser el último.

```
var suma =  function (...numeros)
```

```
{
   let total=0;
   for (let i=0; i< numeros.length; i++)
   {
      total+=parseInt(numeros[i]);
   }

   return (total);
}

console.log(suma());              // mostrará 0
console.log(suma(4.5));           // mostrará 4
console.log(suma(23.5,10));       // mostrará 33
console.log(suma(4.5,5,10,23)); // mostrará 42
```

Hay una tercera forma de declarar una función que es la de funciones **arrow,** que es una forma abreviada de declaración que incorpora varios matices diferenciadores de las declaraciones anteriores. Un matiz tiene que ver con el objeto **this** del que se hablará cuando se vea con un poco más de profundidad sobre las clases e instancias de objetos. Por ahora lo vamos a tomar como una forma abreviada de declarar una función similar a las ya vistas. Otro matiz es que no se dispone del arrray **arguments** en su interior por lo que unicamente podemos utilizar la sintaxis **rest** cuando queramos utilizar un número indeterminado de argumentos. En la declaración de una función **arrow** se omite la palabra reservada **function** y se utiliza el par de caracteres **=>** para separar la lista de argumentos del cuerpo de la función que puede ser un bloque de código encerrado entre llaves o una simple expresión que será el valor devuelto en forma similar a que se hubiera ejecutado un **return** de esa expresión:

Los dos ejemplos anteriores codificados con funciones **arrow** quedarían:

```
var suma = (a,b) => (parseInt(a) + parseInt(b));
/* var suma = (a,b) => {
   return (parseInt(a) + parseInt(b)); // equivalente a la anterior
} */
console.log(suma(23.5,10));  // mostrará 33


var suma =  (...numeros) =>
{
   let total=0;
   for (let i=0; i< numeros.length; i++)
   {
      total+=parseInt(numeros[i]);
   }
   return (total);
}

console.log(suma());  // mostrará 0
console.log(suma(4.5));  // mostrará 4
console.log(suma(23.5,10));  // mostrará 33
console.log(suma(4.5,5,10,23));  // mostrará 42
```

Si la función **arrow** solo tiene un argumento, y no tiene sintaxis **rest**, se pueden omitir los paréntesis:

```
var dobla= numero => (numero * 2);
console.log(dobla(8));  // mostrará 16
```

Ahora que ya sabemos declarar funciones, vamos a modificar el ejemplo en el que se mostraba la fecha actual, para que al hacer clic sobre el texto de la fecha cambie el formato a **dd/mm/aaaa** y cuando se vuelva a hacer clic recupere el primer formato.

```
const divFecha= document.getElementById("divFecha");
var formatoLargo=true;
divFecha.innerHTML = (new Date()).toLocaleDateString("es-ES",
        {weekday:"long",day:"numeric",month:"long",year:"numeric"});
divFecha.addEventListener("click", () => {  //controlador del evento
  formatoLargo = ! formatoLargo;
   if (formatoLargo)
      divFecha.innerHTML = (new Date()).toLocaleDateString("es-ES",
        {weekday:"long",day:"numeric",month:"long",year:"numeric"});
   else
      divFecha.innerHTML = (new Date()).toLocaleDateString("es-ES");
})
```

Con **addEventListener** se añade un controlador de evento, en este caso **click**, al elemento **divFecha**. El controlador se ha escrito en forma de función arrow. La función que actúa como comtrolador del evento puede llevar un argumento que es rellenado de forma automática con información del evento producido, por ejemplo, en este caso si hubiéramos puesto un argumento este sería rellenado con información sobre el **click** como por ejemplo la posición del puntero del ratón, si se pulsó botón derecho o izquierdo, etc.

Puede parecer un error haber declarado la variable **divFecha** como constante ya que su propiedad **innerHTML** va a ser modificada, pero no lo es. En la variable **divFecha** lo que se almacena en realidad es la dirección en la memoria en la que está almacenado el objeto y esta dirección no cambia nunca en este programa por lo tanto es válida la declaración como constante aunque las propiedades del objeto cambien.

Muchos lenguajes de programación, incluido javaScript disponen de un **operador condicional** o **if inline**. Este operador permite devolver uno de dos valores en función de una condición. La sintaxis del operador es;

(*condición*)**?***expresiónSI***:***expresiónNO*

Por ejemplo:

```
numero= a + (a>10)?24:(b+5);
```

numero contendrá el valor de **a** al que se le habrá sumado **24** si el valor de **a** es mayor de **10** o el resultado de sumar **5** al contenido de **b** si fuera menor o igual a **10.** Por ejemplo, si **b** vale **5** y **a** vale **20** en **numero** se almacenará **20+24,** pero si **a** valiera **7** en **numero** se almacenará **7 + 5 + 5.**

El programa anterior se podría haber escrito:

```
const divFecha= document.getElementById("divFecha");
var formatoLargo=true;
```

```
divFecha.innerHTML = (new Date()).toLocaleDateString("es-ES",
{weekday:"long",day:"numeric",month:"long",year:"numeric"});
divFecha.addEventListener("click", () => {
   formatoLargo = ! formatoLargo;
   divFecha.innerHTML = (formatoLargo)?
       (new Date()).toLocaleDateString("es-ES",
{weekday:"long",day:"numeric",month:"long",year:"numeric"}):
       (new Date()).toLocaleDateString("es-ES");
});
```

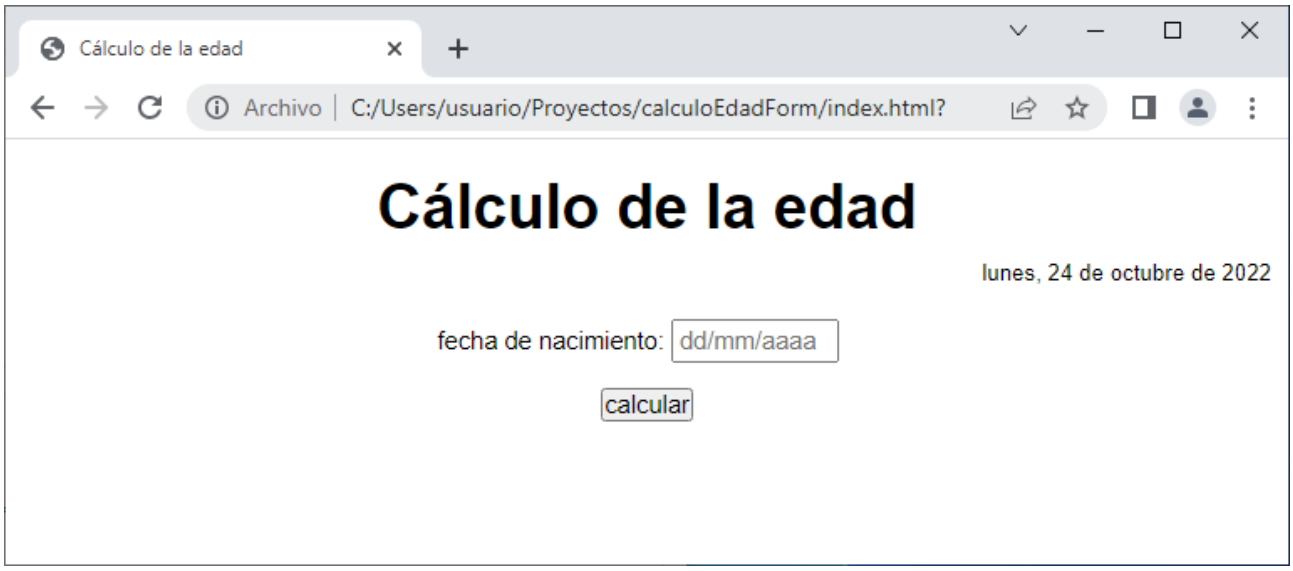
o

```
const divFecha= document.getElementById("divFecha");
var formatoLargo=true;
divFecha.innerHTML = (new Date()).toLocaleDateString("es-ES",
{weekday:"long",day:"numeric",month:"long",year:"numeric"});
divFecha.addEventListener("click", () => {
   formatoLargo = ! formatoLargo;
   divFecha.innerHTML = (new Date()).toLocaleDateString("es-ES",
            (formatoLargo)?
          {weekday:"long",day:"numeric",month:"long",year:"numeric"}:
          {});
});
```

Antes de seguir avanzando en elementos más complejos en javaScript vamos a realizar una serie de ejercicios que nos servirán para aplicar todos los conocimientos que hemos adquirido.

1 - Cálculo de la edad

Vamos a modificar el ejercicio del cálculo de la edad para solicitar la fecha de nacimiento desde un formulario. Se mostrará el resultado sobre la propia página. Al volver a teclear una nueva fecha de nacimiento el resultado del cálculo anterior se borrará. Los mensajes de error también se mostrarán sobre la propia página. Al modificar la fecha de nacimiento que provocó el error también desaparecerá el mensaje. En el código vamos a codificar una función con nombre **edad,** que reciba como argumento una fecha anterior a la fecha actual y devuelva el número de años completos que han transcurrido. Esta función se codificará en una archivo separado para posibilitar su reutilización en otros programas, este archivo se llamará **dateTools.js** y en el iremos añadiendo funciones de utilidad para fechas creadas por nosotros.
La página puede tener la siguiente apariencia:



El código html y css sería:

index.html

```
<!DOCTYPE html>
<html lang="es">

<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Cálculo de la edad</title>
   <link rel="stylesheet" href="css/index.css">
</head>

<body>
   <main>
      <header>
         <h1>Cálculo de la edad</h1>
         <div id="divFecha"></div>
      </header>
      <section>
         <form>
            <div>
               <label for="txtFechaNacimiento">fecha de nacimiento:</label>
               <input type="text" id="txtFechaNacimiento"
                     pattern="(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[0-2])\/((19|20)[0-9]{2})"
                     title="dd/mm/aaaa"
                     maxlength="10"
                     placeholder="dd/mm/aaaa" required>
            </div>
            <div id="divBoton">
               <input type="submit" value="calcular">
            </div>
         </form>
         <div id="divEdad">
            edad: <span id="spnEdad"></span>
```

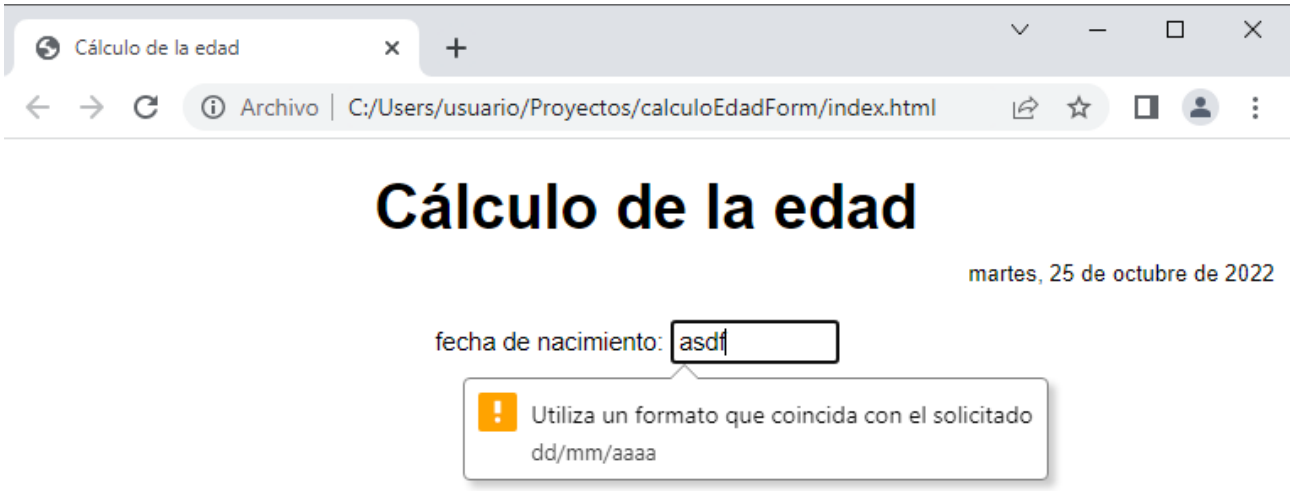
```
            </div>
            <div id="divMensajeError">
            </div>
        </section>
    </main>
    <script src="js/dateTools.js"></script>
    <script src="js/index.js"></script>
</body>
</html>
```

index.css

```
* {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 11pt;
    margin: 0px;
    padding: 0px;
    box-sizing: border-box;
}

body>main>header>h1 {
    font-size: 2.5em;
    text-align: center;
    margin: 20px 0px 10px 0px;
}

body>main>header>div#divFecha {
    font-size: 10pt;
    margin-right: 10px;
    margin-bottom: 20px;
    text-align: right;

}

body>main>section {
    width: 250px;
    margin: auto;
}

body>main>section>form>div>input[type="text"] {
    width: 100px;
    padding: 3px;
}

body>main>section>form>div#divBoton {
    text-align: center;
    margin-top: 15px;
    margin-bottom: 15px;
}

body>main>section>div#divEdad {
    display: none;
    font-size: 1.2em;
    color:blue;
}

body>main>section>div#divMensajeError {
    display: none;
    font-size: 0.85em;
```

```
    color: red;
}
```

En código html, en los formularios, se puede incorporar un cierto control sobre los contenidos que se pueden introducir en las cajas de texto a través del atributo **pattern** y del atributo **type.** En nuestro caso se ha utlizado el atributo **pattern** que contiene una expresión regular que se aplica siempre a todo el contenido de la caja de texto, es decir, es como si tuviese el carácter **^** al inicio de la expresión y el carácter **$** al final. Cuando el contenido de la caja de texto no coincide con lo expresado en la expresión regular, al enviar el formulario se muestra de forma automática un tooltip indicando el error de formato y añade el contenido del atributo **tittle** como se ve en la siguiente imagen.



El formulario se envía al pulsar en el botón **calcular** que es de tipo **submit**. Nosotros en esta página no queremos enviar ningún dato a ningún servidor, todo se hace en la parte cliente, por lo que debemos interceptar el envío y cancelarlo. Lo haremos desde javaScript. No se hacen más comentarios sobre el código html porque se da por supuesto que se tienen suficientes conocimientos de html y css, si no, debería realizarse alguno de los muchos tutoriales que hay en la red antes de seguir con esta iniciación.

En el código html, al final, se ha incorporado la llamada a dos archivos de javaScript. Se colocan **siempre al final** para estar seguros de que las referencias en el código javaScript a elementos del **DOM** ya existan. Si se colocan en el **head** se puede dar el caso de que haga una referencia a un elemento del DOM que no existe todavía. El navegador va interpretando el código html de forma secuencial según el orden en que está escrito. El orden de los dos archivos javaScript también tiene que ver con esto. En el archivo **index.js** se supone que va a haber alguna referencia a la función **edad** ya mencionada, por lo tanto debe estar declarada antes de su uso por lo que el archivo que contiene su declaración debe anteceder al archivo que la usa.(**NOTA**: en este caso concreto no supondría ningún error si apareciese primero el archivo **index.js** y posteriormente el archivo **dataTools.js** porque la llamada a la función, luego se verá, se realizará en respuesta a una acción del usuario cuando ya todo el código ha sido cargado por el navegador, pero siempre es recomendable cargar primero los archivos de declaración y posteriormente los archivos que los usan para evitar posibles errores de llamada a función no declarada)

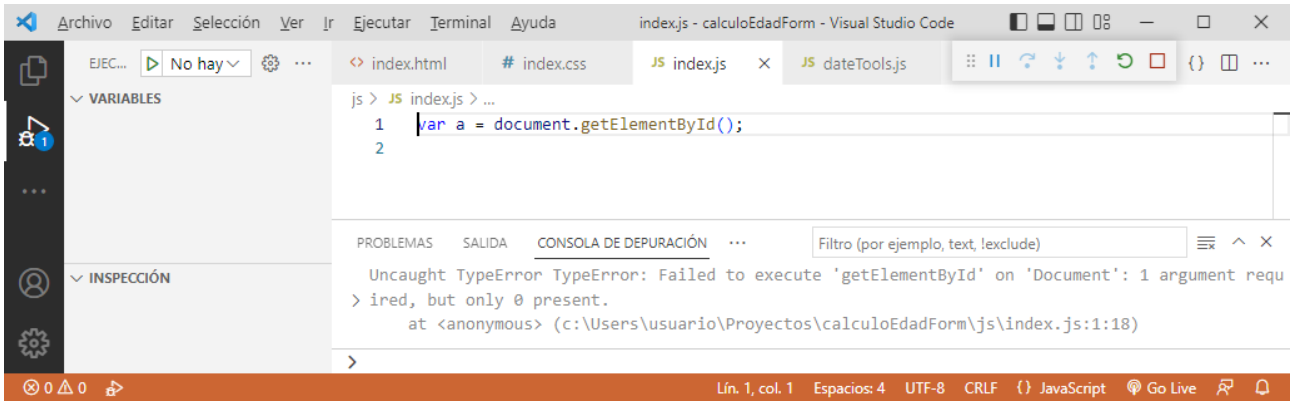
Errores en tiempo de ejecución

Vamos a codificar la función **edad** dentro del archivo **dataTools.js**. La función debe aceptar como argumento un dato de tipo **Date**. ¿Por qué de tipo **Date** y no como una cadena en formato **dd/mm/aaaa** que es lo que introducimos en la caja de texto? Porque siempre que codifiquemos código que va a ser reutilizado lo haremos no solo pensando en la utilización actual, sino en futuros usos. Si fuéramos a utilizar la función en otro programa en donde se solicita por separado el día el mes y el año, tendríamos que recodificar la función, lo mismo que si fuera a ser utilizada en una página en la que el formato fuera **mm/dd/aaaa** que utilizan los anglosajones para las fechas. Otra ventaja de utilizar el tipo **Date** es que siempre habrá una fecha válida en su interior ahorrándonos de comprobar su validez. ¿Cómo se sabe si un argumento es una instancia de una clase de objeto dado? Con el operador **instanceof**.

instancia instanceof *clase*

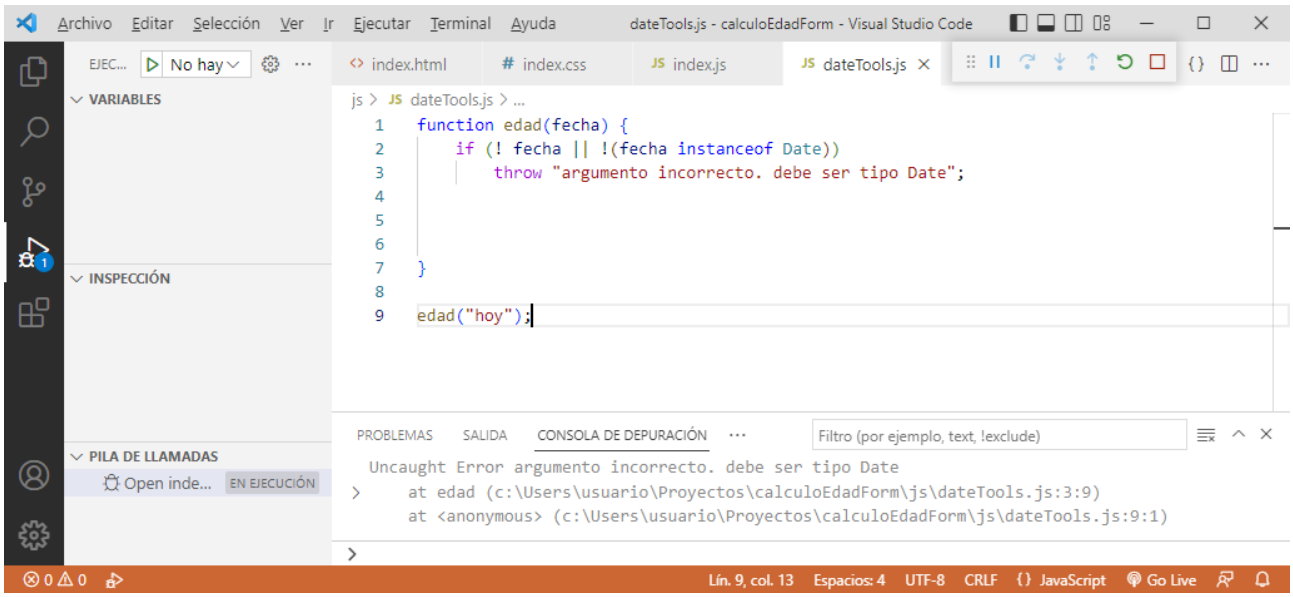
La expresión devuelve **true** o **false**.

Lo primero que haremos en el cuerpo de la función es comprobar que hay un argumento y que es de tipo **Date**. Si sí, continuaremos con el cálculo de la edad, pero si no ¿que se hace? Recordemos que estamos codificando una función que va a ser utilizada en otros programas y en otros entornos, por ejemplo, podría ser utilizada en un programa sin interacción con el usuario para el cálculo de una bonificación por edad y que su fecha de nacimiento se saca de una base de datos. Por lo tanto no siempre hay que sacar en la página un mensaje de error y además este mensaje de erro no tiene por qué ser único. Fijémonos en lo que hacen las funciones que ya están codificadas y son usadas en nuestros programas. Si el argumento no se corresponde con el tipo esperado y puede ser convertido al tipo esperado se hace la conversión y se utiliza como si fuera un valor válido, por ejemplo: **parseInt("23.1")** devuelve **23**. Si el argumento no se corresponde con el tipo y no puede ser convertido, o no existe, o no se corresponde el número de argumentos con el exigido lo que se hace es generar un error en tiempo de ejecución que mostrará en la consola el mensaje de error asociado al mismo.



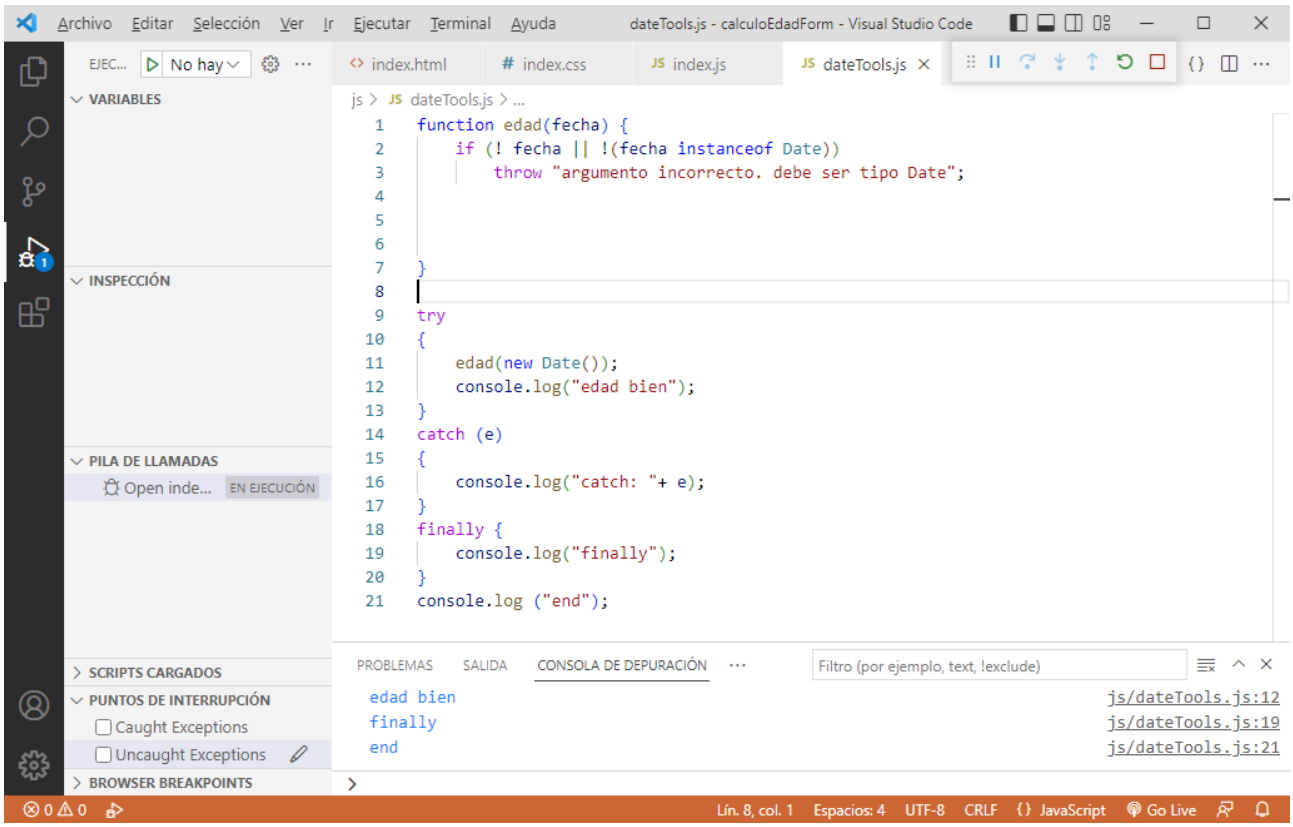
Resumiendo, todas las condiciones anómalas de ejecución de nuestro código reutilizable deben generar un **error en tiempo de ejecución** que comunique al programador esta condición anómala a través de un mensaje. ¿Como se genera un error en tiempo de ejecución al detectar una situación anómala? Haciendo uso de la sentencia **throw** seguida del mensaje de error. En nuestro caso la cabecera de la función quedaría:

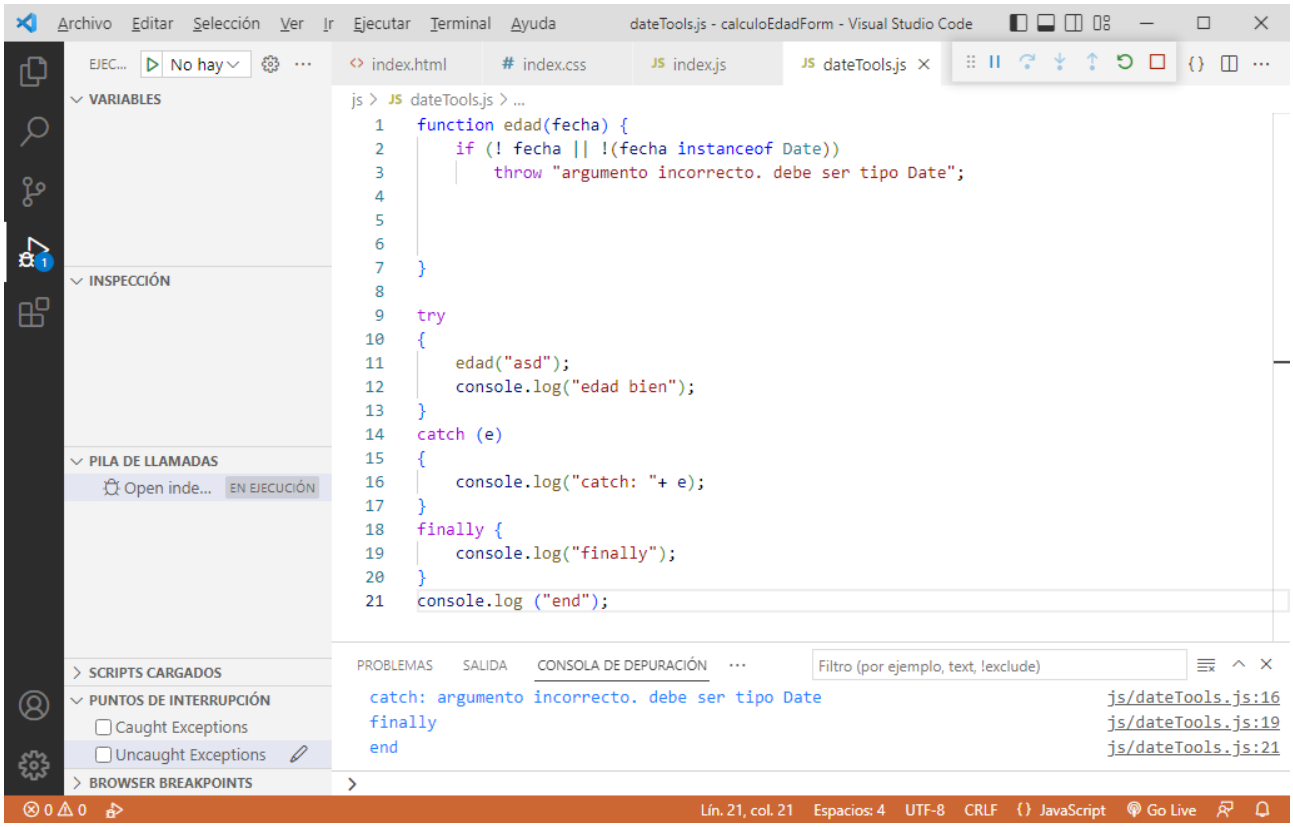
```
function edad(fecha) {
    if (! fecha || !(fecha instanceof Date))
        throw "argumento incorrecto. debe ser tipo Date";
```



Cuando se detecta un error en tiempo de ejecución, la ejecución termina. Sería deseable poder detectar errores en tiempo de ejecución y si es posible continuar la ejecución mostrando por ejemplo un mensaje y dando la oportunidad de solventar el error. Por ejemplo si el usuario introduce una fecha incorrecta, sería deseable que al intentar calcular la edad y desencadenar un error en tiempo de ejecución se mostrara un mensaje indicando que se ha producido un error y que se introduzca una fecha válida. La detección de errores en tiempo de ejecución se hace a través del bloque **try … catch … finally**.

```
try
    {
        sentencias_que pueden_desencadenar_error
    }
catch (excepción)
    {
        sentencias_recuperación_error
    }
[ finally
    {
        sentencias_siempre_se_ejecutan
    } ]
```





Seguimos avanzando con el código de nuestra función **edad**. Ya sabemos que si se superó el primer **if,** nuestro argumento es una fecha, pero esta fecha puede ser posterior a la fecha actual lo cual contradice el requerimiento expuesto. Lo siguiente será comprobar que se cumple y si no se cumple volvemos a desencadenar otro error en tiempo de ejecución, con otro mensaje de error.

Nuestra función, hasta ahora, va a poder desencadenar dos tipos de errores, ambos identificados por una cadena. Para distinguir uno de otro en el bloque **try … catch**, en la parte **catch** podemos interrogar al argumento que va entre paréntesis por su valor de cadena. Otra forma de distinguir ambos errores es desencadenar instancias de errores específicos. Hay una clase de error genérico **Error** y clases específicas, todas ellas hijas de la clase genérica. Así tenemos entre otras, las clases: **TypeError** y **RangeError.**

```
function edad(fecha) {
   if (! fecha || !(fecha instanceof Date))
      throw new TypeError("argumento incorrecto. debe ser tipo Date");
   if (fecha.getTime() >= (new Date()).getTime())
      throw new RangeError("argumento incorrecto. la fecha debe ser anterior al momento actual");
}

try
{
   edad(new Date());
   console.log("edad bien");
}
catch (e)
{
   if (e instanceof TypeError)
```

```
        console.log("error de tipo: "+ e.message);
        if (e instanceof RangeError)
        console.log("error de rango: "+e.message);
}
finally {
    console.log("finally");
}
console.log ("end");
```

En nuestra función estamos en que ya nos hemos cerciorado de que es una fecha y que es anterior a la fecha actual, ya podemos hacer los cálculos como ya teníamos:

```
function edad(fecha) {
    if (! fecha || !(fecha instanceof Date))
        throw new TypeError("argumento incorrecto. debe ser tipo Date");
    let fechaActual=new Date();
    if (fecha.getTime() >= fechaActual.getTime())
        throw new RangeError("argumento incorrecto. la fecha debe ser anterior al momento actual");
    let años = fechaActual.getFullYear() - fecha.getFullYear();
    // si mes y día de la fecha de nacimiento es mayor que mes y día de la fecha actual hay que corregir
en 1
    if (fecha.getMonth() > fechaActual.getMonth() ||
          (fecha.getMonth() ==
         fechaActual.getMonth() && fecha.getDate() >   fechaActual.getDate()))
        años–;
    return años;
}
```

En **index.js**, además de mostrar la fecha actual en **divFecha** tenemos que capturar el evento **submit** para comprobar que la fecha es correcta, hacer el cálculo de la edad y mostrarlo en **spnEdad**. Si se detecta algún error se mostrará el mensaje en **divMensajeError**. Los únicos errores que se pueden detectar aquí son los no detectados por el **pattern**, es decir, que se ponga por ejemplo un 31 de abril, o que se ponga una fecha posterior a la actual. El primero lo detectaremos al crear la instancia para almacenar la fecha introducida en la caja de texto y comprobar que sigue siendo la misma y el segundo lo detectará la función **edad** como un error en tiempo de ejecución. Además se debe cancelar el envío del formulario para que no se recargue la página borrando todos los datos que el usuario introdujo. También se debe hacer que al pulsar cualquier tecla en la caja de texto se borre tanto la edad como cualquier mensaje de error que pudiera haber. Esto se hace capturando el evento **keypress**.

Para mostrar la fecha actual en **divFecha:**

```
document.getElementById("divFecha").innerHTML=
    (new Date()).toLocaleDateString("es-ES",
            {weekday:"long",day:"numeric",month:"long",year:"numeric"});
```

Para borrar cualquier mensaje de error y la edad que pudiera haber, capturamos el evento **keypress:**

```
let txtFechaNacimiento = document.getElementById("txtFechaNacimiento");
let spnEdad = document.getElementById("spnEdad");
let divMensajeError = document.getElementById("divMensajeError");

txtFechaNacimiento.addEventListener("keypress",
```

```
    () => {
        divEdad.style.display = "none";
        divMensajeError.style.display = "none";
    }
);
```

Para capturar el evento **submit** del formulario**:**

```
let frmFormulario = document.querySelector("body>main>section>form");

frmFormulario.addEventListener("submit",
    e => {
        e.preventDefault(); // cancelar la acción por defecto que es el envío
        // necesitamos extraer el día, el mes y el año
        let patrón = /^(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[0-2])\/((?:19|20)[0-9]{2})$/
        patrón.test(txtFechaNacimiento.value);
            // no hay posibilidad de que salga false por el pattern
        let datFechaNacimiento = new Date(RegExp.$3, RegExp.$2 - 1, RegExp.$1);
        // comprobar que es correcta
        if ((parseInt(RegExp.$1) + "/" + parseInt(RegExp.$2) + "/" + RegExp.$3) !=
            (datFechaNacimiento.toLocaleDateString("es-ES"))) {
            divMensajeError.innerHTML = "la fecha no es posible";
            divMensajeError.style.display = "block";
        }
        else {
            try {
                spnEdad.innerHTML = edad(datFechaNacimiento);
                divEdad.style.display="block";
            } catch (error) {
                divMensajeError.innerHTML =
                "la fecha de nacimiento no puede ser posterior a la fecha actual";
                divMensajeError.style.display = "block";
            }
        }
    }
);
```

Hemos visto que para seleccionar un elemento del **DOM** lo más sencillo era incluir un atributo **id** en el código **html** y utilizar el método **getElementById** del objeto **document** pero hay otras formas de seleccionar elementos, entre ellos el que aparece en el código, **querySelector,** que permite la selección del primer elemento que coincida con el selector **CSS,** expresado en forma de cadena. En nuestro ejemplo selecciona el primer elemento contenido **body, main, section** que tenga como nombre de elemento html **form** y por lo tanto hemos seleccionado el formulario. El evento **submit** está asociado al formulario y no al botón que desencadena el **submit**. Podíamos haber capturado el evento **click** sobre el botón y cancelarlo también con el mismo efecto. Queriendo cancelar el envío queda más intuitivo capturar el evento **submit**.

En el código se ve como se cancela la acción por defecto de cualquier evento que es ejecutar el método **preventDefault** que pertenece al objeto **event** que se construye de forma automática y se pasa como argumento a la función controladora del evento. En este caso se ha utilizado una función **arrow** y como solo se declara un argumento no hace falta que esté encerrado entre paréntesis.

Si hubiéramos querido limitar a dígitos y slash / los caracteres que el usuario pudiera introducir en la caja de texto lo podríamos haber hecho en el mismo evento **keypress** incluyento un argumento de tipo **event** y accediendo a su propiedad **key** para ver que carácter se pulsó, si no es válido ejecutaríamos **preventDefault**:

```
txtFechaNacimiento.addEventListener("keypress",
   (e) => {
      if (! /[0-9\/]/.test(e.key))
         e.preventDefault();
      else{
      divEdad.style.display = "none";
      divMensajeError.style.display = "none";
      }
   }
);
```

Para incluir el carácter **/** en la expresión regular se ha procedido a escaparlo: **\/**

El mostrar u ocultar los mensajes de error o la fecha se ha hecho accediendo a la propiedad **style.display** de los elementos correspondientes. A través de la propiedad **style** tenemos acceso a todas las propiedades de estilo CSS en línea del elemento de que se trate, por ejemplo si quisiésemos cambiar el tamaño de la fuente accederíamos a la propiedad **style.fontSize** y le asignaríamos un nuevo valor.

El código final de **index.js** quedaría:

```
document.getElementById("divFecha").innerHTML = (new Date()).toLocaleDateString("es-ES",
      { weekday: "long", day: "numeric", month: "long", year: "numeric" });

let txtFechaNacimiento = document.getElementById("txtFechaNacimiento");
let spnEdad = document.getElementById("spnEdad");
let divMensajeError = document.getElementById("divMensajeError");
let frmFormulario = document.querySelector("body>main>section>form");

txtFechaNacimiento.addEventListener("keypress",
   (e) => {
      if (! /[0-9\/]/.test(e.key))
         e.preventDefault();
      else{
      divEdad.style.display = "none";
      divMensajeError.style.display = "none";
      }
   }
);

frmFormulario.addEventListener("submit",
   e => {
      e.preventDefault(); // cancelar la acción por defecto que es el envío
      // necesitamos extraer el día, el mes y el año
      let patrón = /^(0?[1-9]|[12][0-9]|3[01])\/(0?[1-9]|1[0-2])\/((?:19|20)[0-9]{2})$/
      patrón.test(txtFechaNacimiento.value);
            // no hay posibilidad de que salga false por pattern
      let datFechaNacimiento = new Date(RegExp.$3, RegExp.$2 - 1, RegExp.$1);
      // comprobar que es correcta
      if ((parseInt(RegExp.$1) + "/" + parseInt(RegExp.$2) + "/" + RegExp.$3) !=
```

```
            (datFechaNacimiento.toLocaleDateString("es-ES"))) {
            divMensajeError.innerHTML = "la fecha no es posible";
            divMensajeError.style.display = "block";
        }
        else {
            try {
                spnEdad.innerHTML = edad(datFechaNacimiento);
                divEdad.style.display="block";
            } catch (error) {
                divMensajeError.innerHTML =
                            "la fecha de nacimiento no puede ser posterior a la fecha actual";
                divMensajeError.style.display = "block";
            }
        }
    }
);
```

2 - La primitiva

Se trata de un programa que genere de forma aleatoria una combinación de la lotería primitiva, 6 números elegidos de entre los números 1 a 49 sin que puedan aparecer repetidos. La página tendrá una apariencia incial como la que se ve:

Al pulsar sobre el botón **generar** se mostrará la combinación elegida en la caja bajo **combinación** con sus números ordenados en forma ascendente. En la tabla **números** aparecerán destacados los números elegidos como se ve en la imagen siguiente:

Si se pulsa de nuevo en generar se repetirá el proceso.

Vamos a codificar una función **combinacionPrimitiva** en un archivo **lotoTools.js** separado, con vistas a su reutilización en otros programas. Esta función no recibirá argumentos y devolverá un **array** con 6 casillas y en su interior los 6 números elegidos.

```
function combinacionPrimitiva() {
  const tablaNumeros = [];
  const numerosElegidos = [];
  for (let i = 1; i <= 49; i++)     // array con los 49 números elegibles
    tablaNumeros.push(i);
  for (let i = 1; i <= 6; i++) {
    numerosElegidos.push((tablaNumeros.splice(parseInt(Math.random() * tablaNumeros.length), 1))[0]);
  }
  return numerosElegidos.sort((a, b) => parseInt(a) - parseInt(b));
   }
```

El método **push,** de la clase **Array,** añade al final del array una nueva casilla con el valor que se acompaña como argumento. El método **splice**, también de la clase **Array** permite eliminar y/o añadir elementos a un array. En el código se elige al azar una casilla de entre las que actualmente tiene el array **tablaNumeros** y se borra **1** casilla: (tablaNumeros.**splice**(parseInt(**Math.random()** * tablaNumeros.**length**), 1). El método **Math.random()** devuelve un número al azar comprendido entre **0** y el **1** aunque nunca llega a este, si lo multiplicamos por la longitud del array no generara un número entre **0** y casi la longitud del array. Al tomar su parte enteera nos devolverá un número comprendido entre **0** y uno menos que la longitud del array que se corresponde con un índice válido dentro del array que van de **0** a **length - 1**. El método **splice** devuelve el subarray eliminado, como se ha eliminado una casilla, devuelve un array con una única casilla, de ahí el **[0]** del final. Resumiendo se elige una casilla al azar, se elimina, y se añade su contenido a **numerosElegidos**.

La última sentencia de la función ordena el contenido del array haciendo uso del método **sort** de la clase **Array**. Este método, si se invoca sin argumentos, hace una ordenación alfanumérica de los contenidos, es decir, hace una invocación del método **toString** de cada casilla para hacer la comparación entre contenidos. Si al método se acompaña una función utilizará esta para determinar si dos casillas, representadas por los argumentos **a** y **b** en el código, están ordenadas en la tabla o no, de forma que si la función devuelve un valor positivo es que las dos casillas están desordenadas y se procederá a su intercambio en el array. Si devuelve un valor negativo es que están ordenadas y no se intercambiarán, y si devuelve **0** es que son iguales. En el código se devuelve la resta el valor numérico de la casilla **a** con el de la casilla **b**. Si esta resta es positiva, **a** es mayor que **b**, están desordenados y se intercambiarán, es decir, estamos haciendo una ordenación en ascendente. Si hubiéramos puesto: **parseInt(b) - parseInt(a)** estaríamos haciendo una ordenación en ascendente.

El código html de la página podría ser:

index.html

```
<!DOCTYPE html>
<html lang="es">

<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Lotería primitiva</title>
   <link rel="stylesheet" href="css/index.css">
</head>

<body>
   <main>
      <header>
         <h1>Lotería primitiva</h1>
         <div id="divFecha"></div>
      </header>
      <section>
         <h1>combinación</h1>
         <div id="divCombinación"></div>
         <div class="generar">
            <button id="btnGenerar">generar</button>
         </div>
```

```
        </section>
        <aside>
            <h1>números</h1>
            <div id="divNumeros"></div>
        </aside>
    </main>
    <script src="js/lotoTools.js"></script>
    <script src="js/index.js"></script>
</body>

</html>
```

index.css

```
* {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 12pt;
    margin: 0px;
    padding: 0px;
    box-sizing: border-box;
}

body {
    margin: 15px;
}

body>main>header>h1 {
    font-size: 2em;
    margin-bottom: 20px;
}

body>main>header>div#divFecha {
    font-size: 0.75em;
    padding-right: 15px;
    margin-bottom: 20px;
    text-align: right;
}

body>main>section {
    width: 350px;

}


body>main>section>h1 {
    font-size: 1.5em;
    color: darkgrey;
}


body > main > section > div#divCombinación {
    font-size: 1.3em;
    font-weight: bold;
    padding: 5px;
    width: 95%;
    height: 34px;
    line-height: 24px;
    border: solid 1px darkgrey;
    border-radius: 3px;
```

```
    text-align: center;
    margin-bottom: 15px;
}


body > main > section > div.generar {
    width: 100%;
    text-align: center;
    margin-bottom: 20px;
}

body>main>aside {
    width: 350px;
}

body>main>aside > h1 {
    font-size: 1.15em;
    color: darkgrey;
    margin-bottom: 10px;
}

body>main>aside>div#divNumeros>div {
    float: left;
    width: 30px;
    height: 30px;
    border: solid 1px black;
    border-radius: 3px;
    text-align: right;
    padding-right: 3px;
    font-size: 1.3em;
    line-height: 30px;
    margin: 3px;
    color: rgb(96, 147, 179);
}

body>main>aside>div#divNumeros>div.sinBorde {
    border: solid 1px white;
}

body>main>aside>div#divNumeros>div.destacado {
    background-color: rgb(96, 147, 179);
    color: black;
}
```

En el código javaScript capturaremos el evento **click** sobre el botón, restauraremos la tabla de números a su estado inicial para quitar la combinación que pudiera estar de un click anterior, elegiremos una nueva combinación y resaltaremos los números elegidos en la tabla de números. El resaltar un número de la tabla de números se hace poniendo su atributo htaml **class** a **destacado**. Como la palabra **class** en javaScript es una palabra reservada el acceso a este atributo en código se hace a través de la propiedad **className** del elemento de que se trate.

El código en **index.js** podría quedar:

```
document.getElementById("divFecha").innerHTML =
    (new Date()).toLocaleDateString("es-ES",
        { weekday: "long", day: "numeric", month: "long", year: "numeric" });
```

```
function generarNumeros() {
   let strHTML = "<div class=sinBorde></div>";
   for (let i = 1; (i <= 49); i++) {
      strHTML += `<div>${i}</div>\n`;
   }
   return strHTML;
}


document.getElementById("divNumeros").innerHTML = generarNumeros();

document.getElementById("btnGenerar").addEventListener("click",
   () => {
      const numerosElegidos = combinacionPrimitiva();
      document.getElementById("divNumeros").innerHTML = generarNumeros();
      document.getElementById("divCombinación").innerHTML = numerosElegidos;
      const divNumeros = document.querySelectorAll("body>main>aside>div#divNumeros>div");
      for (let i of numerosElegidos) {
         divNumeros[i].className = "destacado";
      }
   }
)
```

La tabla visible de los números que son elegibles **no** es de dos dimensiones como pudiera parecer. Si la hubiéramos definido con el tag **table** y dos dimensiones complicaría mucho el código. Por lo que se ha optado es por una secuencia de elementos **div** y jugando con **width** en css hacemos que tenga apariencia de una tabla de dos dimensiones. La creación de esta secuencia de **div** se realiza en la función **generaNumeros**. Esta función devuelve una cadena con el código html de la secuencia que posteriormente es asignada a la propiedad **innerHTML** del elemento **divNumeros**.

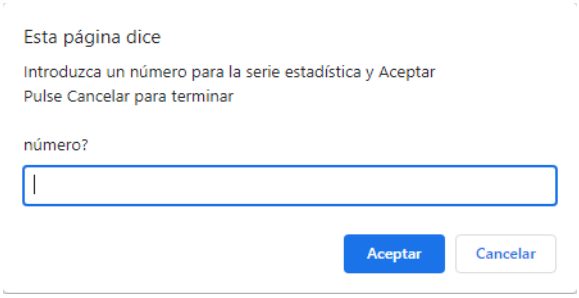
El array con la combinación elegida a través de **combinacionPrimitiva** se almacena en **numerosElegidos**. Vemos que al **innerHTML** de **divCombinación** se le asigna directamente el array **numerosElegidos**. Parece un error, pero no es tal. La propiedad **innerHTML** admite una cadena, como lo que se intenta asignar es un array, se intenta hacer un cambio automático de tipos, ¿como?, ejecutando el método **toString** que tienen todos los objetos de javaScript. ¿Qué hace el método **toString** en un array? Devuelve una cadena con el contenido en forma de cadena de cada casilla separados por coma que es lo que se quería.

Se ve en el código un nuevo formato de la sentencia **for: for (let i of numerosElegidos)** Con este formato se recorre el array **numerosElegidos** y la variable **i** va tomado el valor contenido en cada una de las casillas del array. Si en lugar de la palabra **of** hubiéramos puesto la palabra **in** recorreríamos igualmente el array pero la variable **i** iría tomando el valor de cada índice de cada casilla.

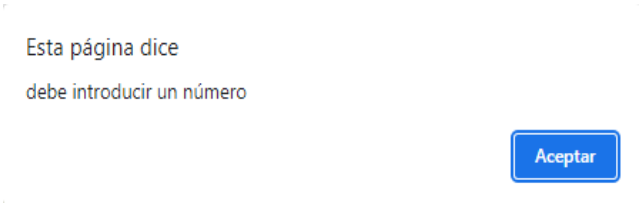
En el array **divNumeros** se almacenan las referencias a todos los elementos **div** contenidos en el elemento html **divNumeros**. Se ha utilizado el método **querySelectorAll** que devuelve un array con todas las referencias que cumplan con el selector css puesto como argumento al método. Habíamos utilizado ya el método **querySelector** que devolvía la primera de estas referencias.

3 - Datos estadísticos

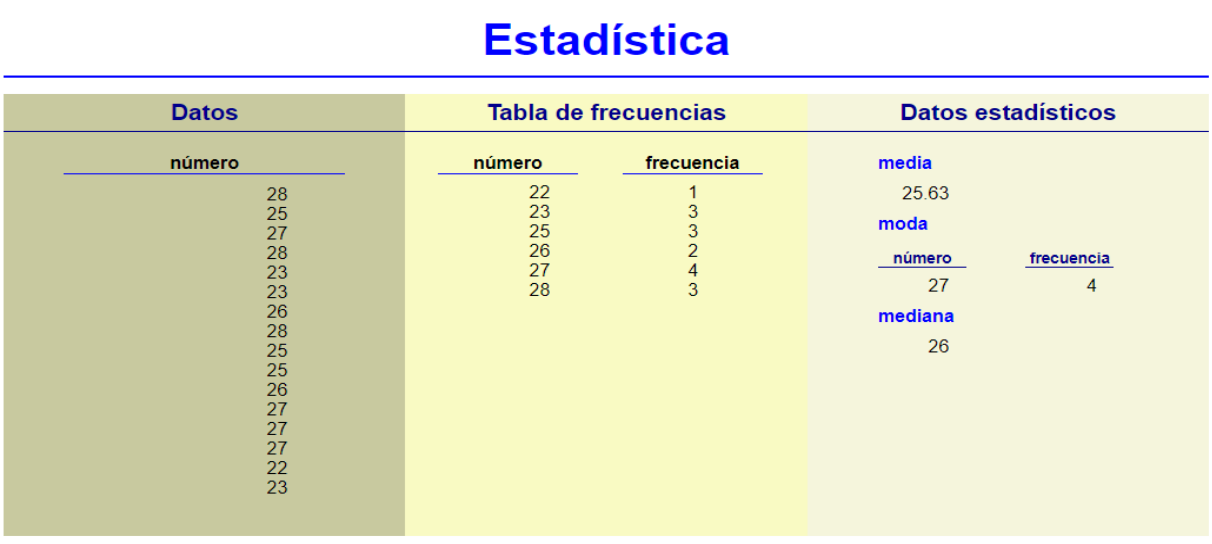
Se quiere codificar una página web que realice el cálculo de los valores estadísticos, **media, moda y mediana**, de una serie de datos numéricos. Los datos se solicitarán al inicio de la carga de la página mediante **prompt:**



Si se introduce un dato que no sea numérico se mostrará el siguiente mensaje, mediante una caja **alert,** y se volverá a pedir de nuevo otro número. Si se da **Aceptar** y no hay nada en número no se mostrará ningún mensaje volviendo a solicitar de nuevo otro número. Los números pueden ser de cualquier tipo, enteros o float, positivos o negativos:



La introducción de la serie de datos quedará terminada cuando se pulse el botón **Cancelar**, pasando a mostrar la página con la siguiente apariencia:



Está desarrollada para un ancho de **1024px**. No es adaptativa y a resoluciones superiores aparecerá centrada. A resoluciones inferiores mostrará barras de scroll.

En la primera columna de la página, **Datos,** se mostrarán los datos según el orden de introducción, es decir, sin ordenar ni agrupar.

En la columna segunda, **Tabla de frecuencias,** se mostrarán los datos distintos introducidos y el número de veces que están repetidos, ordenados en ascendente por **número.** Y en la tercera columna se mostrarán los **datos estadísticos** de la serie introducida.

Los altos de las columnas serán variables en función de su contenido.

Se deben codificar las siguientes funciones:

1. **TablaFrecuencias**. Admitirá un número variable de argumentos, todos numéricos, y devolverá un **array con índices numéricos**, en el que cada casilla es un **objeto** con una propiedad **numero** y una propiedad **frecuencia** que contendrá respectivamente cada número de la serie de argumentos distinto y el número de veces que aparece repetido en la misma. Si alguno de los argumentos no fuera numérico se generará un **error en tiempo de ejecución** con el mensaje: **el argumento xx no es numérico**. Siendo **xx** el argumento de que se trate. Si la lista de argumentos está vacía devolverá un array vacío. El array se devolverá ordenado por **numero** de cada casilla en ascendente. Esta tabla es la que se mostrará en la página en la columna **Tabla de frecuencias**.
2. **Media**. Admitirá un número variable de argumentos, todos numéricos, y devolverá el valor **medio** de los mismos, con dos decimales. Si alguno de los argumentos no fuera numérico se generará un **error en tiempo de ejecución** con el mensaje: **el argumento xx no es numérico**. Siendo **xx** el argumento de que se trate. Si la lista de argumentos está vacía devolverá **null**
3. **Moda**. Admitirá un número variable de argumentos, todos numéricos, y devolverá un array con el/los valores **moda** y sus frecuencias. Cada casilla es un objeto con propiedad **numero** y **frecuencia.** Si alguno de los argumentos no fuera numérico se generará un **error en tiempo de ejecución** con el mensaje: **el argumento xx no es numérico**. Siendo **xx** el argumento de que se trate. Si la lista de argumentos está vacía devolverá **un array vacío.**
4. **Mediana**. Admitirá un número variable de argumentos, todos numéricos, y devolverá el valor **mediana** de los mismos, con dos decimales. Si alguno de los argumentos no fuera numérico se generará un **error en tiempo de ejecución** con el mensaje: **el argumento xx no es numérico**. Siendo **xx** el argumento de que se trate. Si la lista de argumentos está vacía devolverá **null**

Estas funciones se codificarán en archivo **js** separado y con nombre **estadisticas.js.** En este archivo **solo** estarán estas funciones.

estadistica.html

```
<!DOCTYPE html>
<html lang="en">

<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Estadística</title>
   <link href="css/estadistica.css" rel="stylesheet" type="text/css" />
</head>
```

```
<body>
  <main>
    <header>
      <h1>Estadística</h1>
    </header>
    <section>
      <header>
        <h1>Datos</h1>
        <h2>número</h2>
      </header>
      <div id="divNumeros">

      </div>
    </section>
    <section>
      <header>
        <h1>Tabla de frecuencias</h1>
        <h2>número</h2>
        <h2>frecuencia</h2>
      </header>
      <div id="divTablaFrecuencias">

      </div>
    </section>
    <section>
      <header>
        <h1>Datos estadísticos</h1>
      </header>
      <div id="divDatosEstadisticos">

      </div>
    </section>
  </main>

  <script src="js/estadisticaLib.js" type="text/javascript"></script>
  <script src="js/estadistica.js" type="text/javascript"></script>
</body>

</html>
```

estadistica.css

```
* {
  font-family: Arial, "Nimbus Sans L", sans-serif;
  font-size: 12pt;
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body > main {
  width: 1024px;
  margin: auto;
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
body > main > header {
  grid-column: 1/4;
```

```
  grid-row: 1;
  margin-bottom: 15px;
}
body > main > header > h1 {
  width: 100%;
  margin-top: 25px;
  text-align: center;
  padding-bottom: 10px;
  border-bottom: solid 2px blue;
  color: blue;
  font-size: 2.5em;
}
body > main > section > header > h1 {
  font-size: 1.3em;
  color: darkblue;
  padding: 5px;
  text-align: center;
  border-bottom: solid 1px darkblue;
}
body > main > section > header > h2 {
  width: 70%;
  text-align: center;
  border-bottom: solid 1px blue;
  margin-left: 15%;
  margin-right: 15%;
  padding-top: 20px;
}
body > main > section:nth-of-type(0) {
  margin-top: 15px;
  grid-row: 2;
  grid-column: 1/4;
}
body > main > section:nth-of-type(1) {
  grid-row: 3;
  grid-column: 1;
  background-color: #c8c99f;
  padding-bottom: 25px;
}
body > main > section:nth-of-type(2) {
  grid-row: 3;
  grid-column: 2;
  background-color: #f9fac3;
  padding-bottom: 25px;
}
body > main > section:nth-of-type(2) > header h2 {
  width: 35%;
  float: left;
  margin-left: 8%;
  margin-right: 3%;
  margin-bottom: 8px;
}
body > main > section:nth-of-type(3) {
  grid-row: 3;
  grid-column: 3;
  background-color: beige;
  padding-bottom: 25px;
}
body > main > section > div {
  width: 100%;
```

```
  padding: 10px;
  padding-right: 80px;
}
body > main > section > div#divNumeros {
  text-align: right;
  padding-right: 100px;
}
body > main > section > div#divTablaFrecuencias > div {
  width: 37%;
  float: left;
  margin-left: 8%;
  margin-right: 5%;
  text-align: right;
}
body > main > section > div#divDatosEstadisticos h2 {
  width: 100%;
  color: blue;
  float: left;
  margin-left: 8%;
  padding-left: 30px;
  margin-top: 10px;
  margin-bottom: 10px;
}
body > main > section > div#divDatosEstadisticos h3 {
  color: darkblue;
  font-size: 0.85em;
  border-bottom: solid 1px darkblue;
  width: 30%;
  text-align: center;
  float: left;
  margin-left: 50px;
  margin-top: 5px;
  margin-bottom: 8px;
}
body > main > section > div#divDatosEstadisticos > div {
  width: 50%;
  text-align: right;
  float: left;
  padding-right: 15px;
```

estadistica.js

```
// solicitar datos
var aDatos=[];
var oNumeros=document.getElementById("divNumeros");
// bucle infinito, se sale con break
while (true)
{
   let numNumero= window.prompt("Introduzca un número para la serie estadística y Aceptar\nPulse
Cancelar para terminar\n\nnúmero? ");
   if (numNumero==null)
      break;
   if (isNaN(numNumero))
      alert("debe introducir un número");
   else if (numNumero=="")
         continue;
      else
         {
            aDatos.push(numNumero);
```

```
            oNumeros.innerHTML+="<div>"+numNumero+"</div>";
        };
};

// cálculo de frecuencias
var strTablaFrecuencias="";
for (let entrada of TablaFrecuencias(...aDatos))
{
   strTablaFrecuencias+="<div>"+entrada.valor+"</div><div>"+entrada.frecuencia+"</div>";
}
document.getElementById("divTablaFrecuencias").innerHTML=strTablaFrecuencias;

//cálculo de la media
var strDatosEstadisticos="<h2>media</h2><div>"+Media(...aDatos)+"</div>";

//cálculo de la moda
strDatosEstadisticos+="<h2>moda</h2>\n<h3>número</h3>\n<h3>frecuencia</h3>";
for (let entrada of Moda(...aDatos))
{
   strDatosEstadisticos+="<div>"+entrada.valor+"</div><div>"+entrada.frecuencia+"</div>";
}
//cálculo de la mediana
strDatosEstadisticos+="<h2>mediana</h2><div>"+Mediana(...aDatos)+"</div>";

document.getElementById("divDatosEstadisticos").innerHTML=strDatosEstadisticos;
```

estadisticaLib.js

```
function Media(...datos)
{
   // comprobar que hay argumentos
   if (datos==undefined)
      return null;
    // comprobar que son numéricos
   for (let valor of datos)
      // comprobar que son número
      if (isNaN(valor))
         throw new TypeError(`el argumento ${valor} no es numérico`);
   return (datos.reduce( function(suma,valor){
     return  parseFloat(suma)+parseFloat(valor);
   })/datos.length).toFixed(2);
}

function TablaFrecuencias (...datos)
{
   // comprobar que hay argumentos
   if (datos==undefined)
      return null;
    // comprobar que son numéricos
    for (let valor of datos)
    // comprobar que son número
    if (isNaN(valor))
       throw new TypeError(`el argumento ${valor} no es numérico`);

   let aValoresYFrecuencias=[];
   datos.forEach( function (valor)
   {
      let intIndiceValor=aValoresYFrecuencias.findIndex(
            function(valorEnValoresYFrecuencias) {
```

```
                return (valorEnValoresYFrecuencias.valor == parseFloat(valor));
            });
        if (intIndiceValor== -1)
            aValoresYFrecuencias.push({valor:parseFloat(valor),frecuencia:1});
        else
            aValoresYFrecuencias[intIndiceValor].frecuencia++;
    });
    return aValoresYFrecuencias.sort(
            function(a,b){
                return a.valor-b.valor;
            });
}

function Moda(...datos)
{
    // comprobar que hay argumentos
    if (datos==undefined)
        return null;
     // comprobar que son numéricos
     for (let valor of datos)
     // comprobar que son número
     if (isNaN(valor))
         throw new TypeError(`el argumento ${valor} no es numérico`);

    let aModa;
    let aValoresYFrecuencias=TablaFrecuencias(...datos);
    aValoresYFrecuencias.forEach(
        function(entrada,indice)
        {
            if ((! aModa) || (entrada.frecuencia > aModa[0].frecuencia))
                aModa=Array.of({valor:entrada.valor,frecuencia:entrada.frecuencia});
            else if (entrada.frecuencia == aModa[0].frecuencia)
                    aModa.push({valor:entrada.valor,frecuencia:entrada.frecuencia});
        });
    return aModa.sort(function (a,b)
    {
        return a.valor-b.valor;
    });
}

function Mediana(...datos)
{
    // comprobar que hay argumentos
    if (datos==undefined)
        return null;
     // comprobar que son numéricos
     for (let valor of datos)
     // comprobar que son número
     if (isNaN(valor))
         throw new TypeError(`el argumento ${valor} no es numérico`);

    var aAux= Array.from(datos).sort(
            function(a,b)
            {
                return parseFloat(a)-parseFloat(b);
            });
    return (aAux.length % 2 != 0)?aAux[parseInt(aAux.length/2)]:(parseFloat(aAux[aAux.length/2])
+parseFloat(aAux[aAux.length/2-1]))/2;
}
```

En este programa aparecen algunos elementos nuevos: en css el posicionamiento mediante **grid** y en javaScript de algunos de los muchos métodos de los que dispone la clase **Array.**

```
// bucle infinito, se sale con break
while (true)
{
   let numNumero= window.prompt("Introduzca un número para la serie estadística y Aceptar\nPulse
Cancelar para terminar\n\nnúmero? ");
   if (numNumero==null)
      break;
   if (isNaN(numNumero))
      alert("debe introducir un número");
   else if (numNumero=="")
         continue;
      else
         {
            aDatos.push(numNumero);
            oNumeros.innerHTML+="<div>"+numNumero+"</div>";
         };
};
```

El anterior código solicita por **prompt** un número indeterminado de datos. Hay un bucle infinito: **while (true)** del que se sale mediante **break** cuando el usuario pulsa el botón *Cancelar* quedando almacenado en la variable **numNumero** el valor **null**. La sentencia **continue** hace que se deje de ejecutar lo que queda del bucle volviendo la ejecución a la sentencia **while** para que la condición vuelva a ser evaluada. Si nos fijamos esta sentencia en este código es innecesaria si hubiéramos codificado:

```
// bucle infinito, se sale con break
while (true)
{
   let numNumero= window.prompt("Introduzca un número para la serie estadística y Aceptar\nPulse
Cancelar para terminar\n\nnúmero? ");
   if (numNumero==null)
      break;
   if (isNaN(numNumero))
      alert("debe introducir un número");
   else if (numNumero!="")
         {
            aDatos.push(numNumero);
            oNumeros.innerHTML+="<div>"+numNumero+"</div>";
         };
};
```

En lugar de un bucle infinito y **break** podíamos haber utilizado una variable switch:

```
// variable interruptor
let seguir = true;
while (seguir) {
   let numNumero = window.prompt("Introduzca un número para la serie estadística y Aceptar\nPulse
Cancelar para terminar\n\nnúmero? ");
   if (numNumero == null)
      seguir = false;
   else {
      if (isNaN(numNumero))
```

```
            alert("debe introducir un número");
        else if (numNumero != "") {
            aDatos.push(numNumero);
            oNumeros.innerHTML += "<div>" + numNumero + "</div>";
        };
    }
};
```

El método **push** añade una nueva casilla al array.

```
function Media(...datos)
{
    // comprobar que hay argumentos
    if (datos==undefined)
        return null;
     // comprobar que son numéricos
    for (let valor of datos)
        // comprobar que son número
        if (isNaN(valor))
            throw new TypeError(`el argumento ${valor} no es numérico`);
    return (datos.reduce( function(suma,valor){
      return  parseFloat(suma)+parseFloat(valor);
    })/datos.length).toFixed(2);
}
```

Para recordar, el uso de los argumentos número indeterminado con sintaxis **rest**: **...datos**, el formato del bucle **for...of**, y el lanzamiento de un error en tiempo de ejecución con **throw**.
Se hace uso del método **reduce** de la clase **Array** para calcular la suma de todos sus elementos. Este método tiene como argumento una función que es invocada con cada elementos del array. Esta función tiene como primer argumento una variable que va a servir de **acumulador** dentro de la función. El valor devuelto por la función es asignado a esta variable acumulador. El segundo argumento nos sirve de referencia al **valor** de la casilla a la que en un determinado momento se está accediendo. La llamada al método anterior sería equivalente al siguiente código:

```
let suma=0;
for (let valor of datos)
    suma+=parseFloat(valor);
return ((suma/datos.length).toFixed(2));
```

La función que calcula la tabla de frecuencias:

```
function TablaFrecuencias (...datos)
{
    // comprobar que hay argumentos
    if (datos==undefined)
        return null;
     // comprobar que son numéricos
     for (let valor of datos)
     // comprobar que son número
     if (isNaN(valor))
         throw new TypeError(`el argumento ${valor} no es numérico`);

    let aValoresYFrecuencias=[];
    datos.forEach( function (valor)
    {
        let intIndiceValor=aValoresYFrecuencias.findIndex(
```

```
            function(valorEnValoresYFrecuencias) {
                return (valorEnValoresYFrecuencias.valor == parseFloat(valor));
            });
        if (intIndiceValor== -1)
            aValoresYFrecuencias.push({valor:parseFloat(valor),frecuencia:1});
        else
            aValoresYFrecuencias[intIndiceValor].frecuencia++;
    });
    return aValoresYFrecuencias.sort(
            function(a,b){
                return a.valor-b.valor;
            });
}
```

En esta función encontramos el método **forEach** de la clase **Array**. Este método admite un argumento que es una función que es invocada para cada casilla del array. Esta función que es llamada admite un argumento que es la casilla que es accedida en cada momento. Vemos que varios métodos de la clase **Array** admiten un argumento que es una función que es ejecutada para cada casilla del array, se denominan de forma genérica funciones de **callback**. En el ejemplo, para cada casilla del array **datos** se busca su valor en la tabla **aValoresYFrecuencias**. Cada casilla de esta tabla es un objeto con dos propiedades **valor y frecuencia**. Para buscar el valor de la casilla en este array se hace uso del método **findIndex** que, como se ve, admite una función de **callback**. Si esta función devuelve **true** es que la búsqueda ha terminado y devuelve el índice de la casilla en la que se encontró (aquella casilla en la que la función devolvió **true**). **FindIndex** devolverá **-1** si en ninguna casilla de la tabla que se recorre la función de **callback** devuelve **true**. Si es así, se añade una nueva casilla al array **aValoresYFrecuencias** con un nuevo objeto con las dos propiedades. Si la búsqueda tiene éxito se incrementa en **1** su **frecuencia**. Finalmente se orden el array de **aValoresYFrecuencias** en orden ascendente de **valor** mediante el método **sort.**

En el cálculo de la moda:

```
function Moda(...datos)
{
    // comprobar que hay argumentos
    if (datos==undefined)
        return null;
     // comprobar que son numéricos
     for (let valor of datos)
     // comprobar que son número
     if (isNaN(valor))
         throw new TypeError(`el argumento ${valor} no es numérico`);

    let aModa;
    let aValoresYFrecuencias=TablaFrecuencias(...datos);
    aValoresYFrecuencias.forEach(
        function(entrada,indice)
        {
            if ((! aModa) || (entrada.frecuencia > aModa[0].frecuencia))
                aModa=Array.of({valor:entrada.valor,frecuencia:entrada.frecuencia});
            else if (entrada.frecuencia == aModa[0].frecuencia)
                    aModa.push({valor:entrada.valor,frecuencia:entrada.frecuencia});
        });
    return aModa.sort(function (a,b)
    {
        return a.valor-b.valor;
```

```
    });
}
```

Se vuelve a utilizar el método **forEach,** el método **push** y el método **sort**. Aparece el método **of**. Va precedido del nombre de la clase y no del nombre de una instancia. Hasta ahora hemos hablado de métodos de objetos que son invocados siempre tras haber sido creada una instancia de la clase a la que pertenecen. Existen también métodos que pertenecen a la clase y no a las instancias cuando son creadas, se denominan métodos **estáticos**. El concepto se puede aplicar también a propiedades. Por ejemplo, la clase **String** dispone de un método estático **fromCharCode** que devuelve la cadena que se corresponde con el código de carácter que se acompaña como argumento. **Array.fromCharCode(65)** devuelve una **A.** E invocar **"Salamanca".fromCharCode(65)** daría un error porque el método **fromCharCode** no es un método de la instancia:

```
Uncaught TypeError TypeError: "Salamanca".fromCharCode is not a function
```

Otro ejemplo: la clase **Math** que contiene métodos para cálculos matemáticos, solo tiene métodos y propiedades estáticas. **Math.cos** devuelve el coseno de los grados pasados como argumento en radianes **y Mat.PI** es una propiedad que devuelve el valor de la constante **PI.**

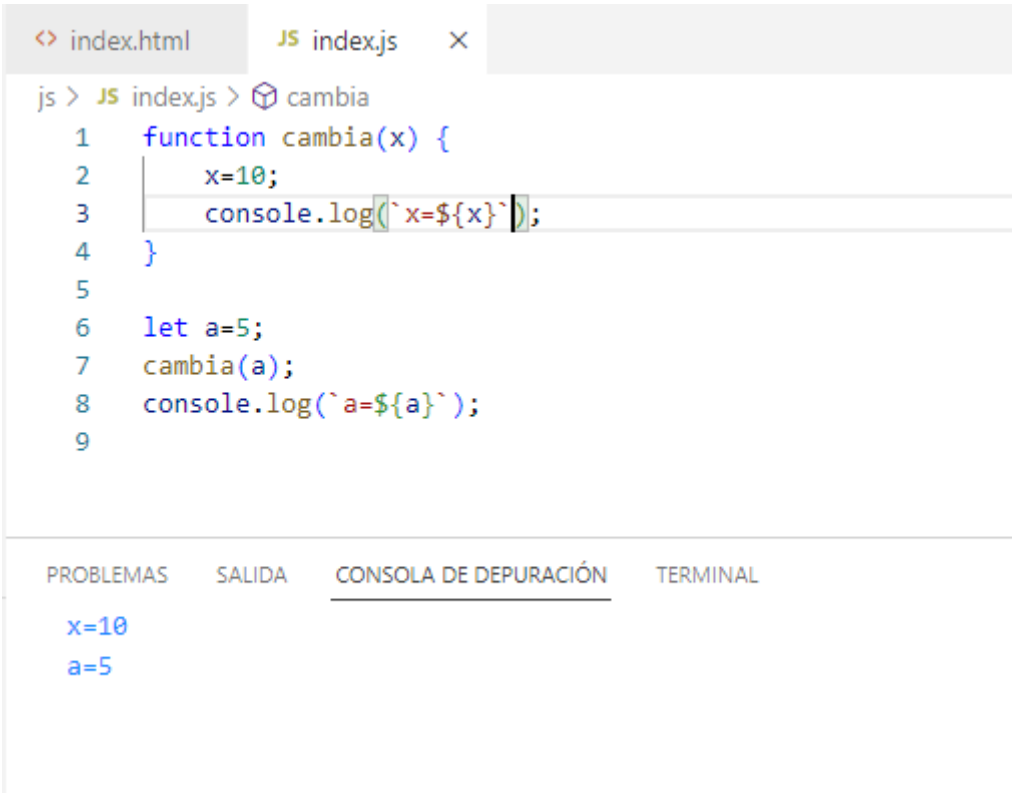
El método estático **of** de la clase **Array** devuelve un nuevo array en el que cada casilla es cada uno de los argumentos pasados al método, es pues, un método alternativo a la creación de arrays que ya habíamos visto: **new Array** o utilización de **[ ]**.

aModa=Array.of({valor:entrada.valor,frecuencia:entrada.frecuencia}); crea un nuevo array con una única casilla que es un objeto con dos propiedades: **valor** y **frecuencia**.

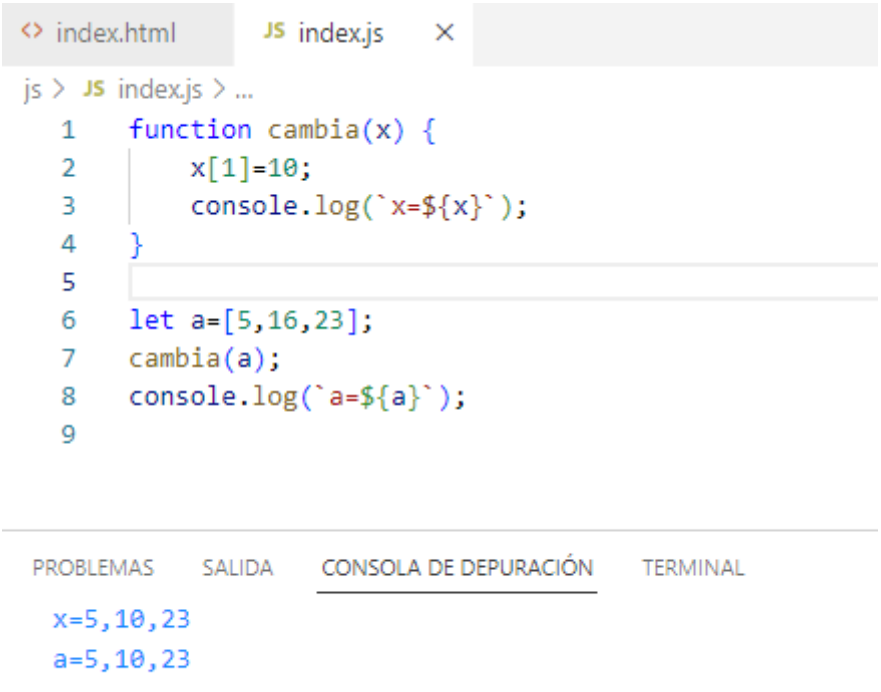
Se utiliza un array para almacenar la moda porque puede haber varios valores entre los datos que sean moda, es decir, su frecuencia sea la misma.

En el cálculo de la mediana no aparece ningún método nuevo de la clase **Array.**

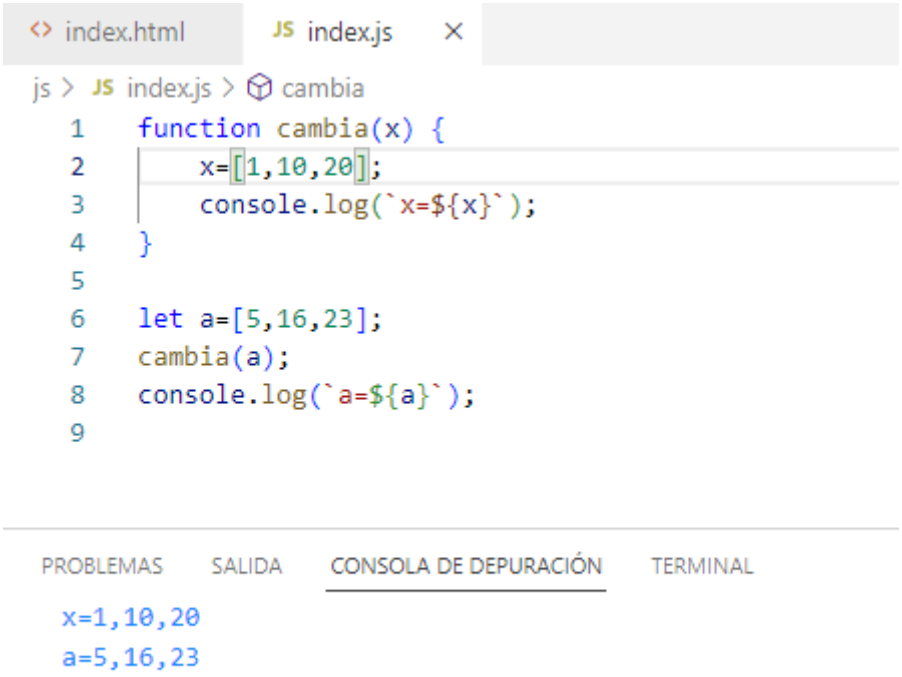
El método **of** es usado frecuentemente para clonar un array. Se debe tener cuidado cuando se pasa como argumento un array, en general cualquier objeto, a una función. Los argumentos en javaScript se dice que se pasan **por valor**, esto significa que si el valor del argumento cambia en la función, este cambio no sale de la función:



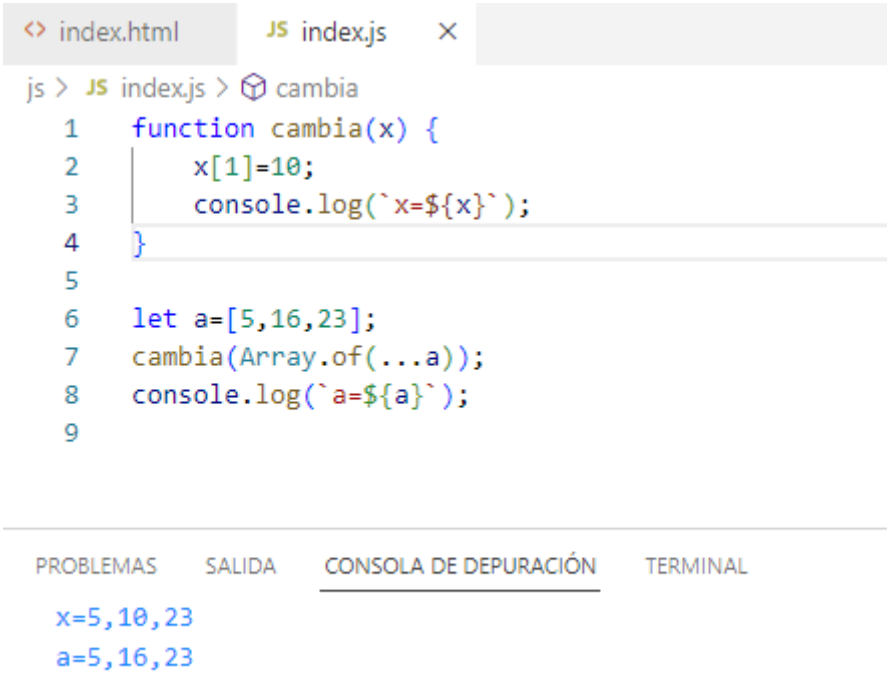
Esto es cierto y puede parecer mentira si vemos la siguiente ejecución:



El contenido de la casilla **1** ha salido fuera de la función y aparentemente contradice lo anteriormente comentado. Cuando se pasa un objeto a otra variable o como argumento de una función o método lo que se pasa es la dirección en la que está alojado el dato, y esta dirección es la que si se cambia no sale de la función pero no impide que lo alojado en esa dirección cambie como se ve en el ejemplo anterior. El siguiente ejemplo lo demuestra:



Dentro de la función se crea un nuevo array, por lo tanto la dirección cambia, y este cambio no sale de la misma. Entonces cuando se pasa un array como argumento a una función y dentro de esta se quiere preservar el array original ante potenciales cambios dentro de la función lo que se hace es trabajar con un clon del array original.



Nótese que si el contenido de alguna casilla del array clonado es de tipo objeto pasará lo mismo con ellos y por lo tanto también se deberán clonar.

Objetos

Ya hemos comentado que en javaScript todo es considerado un objeto, incluso los tipos de datos que no lo son como los tipos de datos primitivos: number, string … que son envueltos (**wrapped**) en objetos equivalentes.

Ya conocemos algunas de las clases que nos ofrece el lenguaje como **Date** o **Array**. También conocemos los objetos que nos ofrece el navegador a través del **DOM** como el objeto **window**. Todos los objetos del **DOM** son instancias de clases que no son accesibles directamente y que nos permiten instanciar y usar en la página. Por ejemplo la clase **HTMLDivElement** es la clase de la que han sido instanciados todos los elementos **DIV** que estén declarados en el html de la página. Estos elementos no son instanciables directamente sino a través del método **createElement** de **document**.

Hasta ahora hemos modificado el contenido de la página web a través de javaScript haciendo uso de la propiedad **innerHTML** del elemento que se quería modificar. La página es construida por el navegador a través de una serie de elementos llamados **nodos,** con una estructura jerárquica de padres e hijos con los elementos especificados en el html. Cada uno de estos nodos se corresponde con elemento o tag del código html. Contenido en el objeto **window** hay un objeto **document** y bajo el todos estos elementos o **nodos**. Ahí nos encontraremos con un objeto **body** y dentro con los objetos contenidos, por ejemplo un objeto **main, article** …. Cada uno de los elementos del **DOM** tiene una propiedad **childNodes** que contiene la lista de todos los elementos

contenidos en él. También se dispone de una serie de métodos para gestionar esta lista. En https://developer.mozilla.org/es/docs/Web/API/Node encontraremos una referencia a los elementos de la clase **Node**. En el ejemplo siguiente se añaden por código varios elementos **div** a un elemento **article** y haciendo uso de los métodos que nos ofrecen los nodos del **DOM.**

index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Prueba</title>
</head>
<body>
   <h1>Haz click para añadir 10 números</h1>
   <article></article>
   <script src="js/index.js"></script>
</body>
</html>
```

index.js

```
var inicio=1;
document.querySelector("h1").addEventListener("click",
   () => {
      const artículo = document.querySelector("article");
      let div;
      for (let i = 1; i <= 10; i++,inicio++) {
         div = document.createElement("div");
         div.innerHTML = `<b>${inicio}</b><br>`;
         artículo.appendChild(div);
      }

   }
)
```

La clase **Node** y por lo tanto cualquier elemento de la página tiene métodos para gestionar esta lista, es decir, añadir, eliminar, modificar, recorrer …

javaScript implementa de forma parcial el paradigma de la programación orientada a objetos, esto quiere decir que no implementa todas las funcionalidades que dicho paradigma define, aunque poco a poco va incorporando más funcionalidades.

Ya tenemos un ligero concepto de lo que son las **clases** y las **instancias**, así como de **propiedades** y **métodos**. También sabemos lo que son los métodos y propiedades **estáticas**. Vamos a ver como se define una nueva clase con sus propiedades y métodos, y también vamos a ver como se declara una propiedad o método estático.

Hay dos formas de declarar una clase nueva. La primera forma es la más antigua y que era la única hasta versiones recientes de javaScript que ya incorporan la declaración de clases a través de la sentencia declarativa **class.** En las versiones anteriores se declaraba una clase no con una

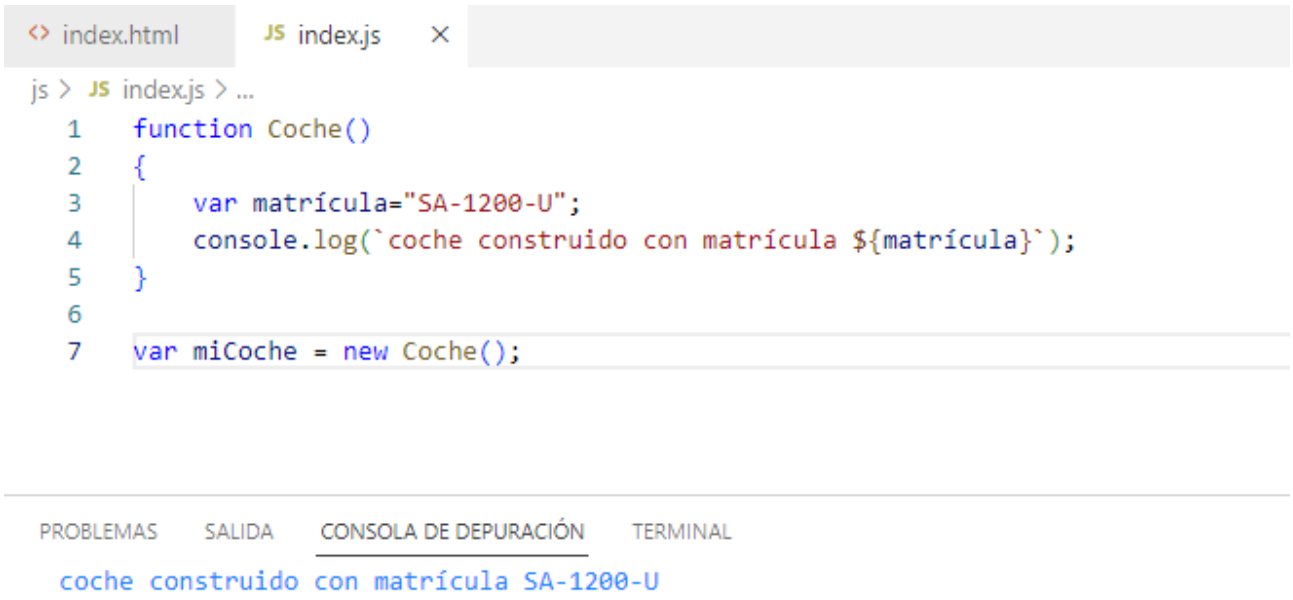
declaración explícita sino a través de la declaración de una **función**. Como todo en javaScript es un objeto, una función es también un objeto, pero además es también un declarador de clases, por lo tanto una declaración como:

```
function Coche()
{
    var matrícula="SA-1200-U";
}
```

Ya declara una clase y además declara una propiedad, pero esta propiedad es privada y no es accesible desde fuera de la clase. Es válido por tanto hacer **let miCoche= new Coche();** Al ser **matrícula** una propiedad privada no está permitido por ejemplo hacer **miCoche.matrícula = "XXXX-ABD";** El acceso a **matrícula** solo está permitido dentro de la declaración de la función. Lo mismo ocurre con los métodos. Un método es una función declarada dentro de la clase, o sea dentro de la función, que actúa como constructora de la clase. Un método **constructor** es aquel que se ejecuta al instanciar la clase. Cada vez que se hace **new Coche()** se ejecuta el código de la función constructora es decir el código de la función **Coche**. Así por ejemplo si la función **Coche** hubiera contenido:

```
function Coche()
{
    var matrícula="SA-1200-U";
    console.log(`coche construido con matrícula ${matrícula}`);
}
```

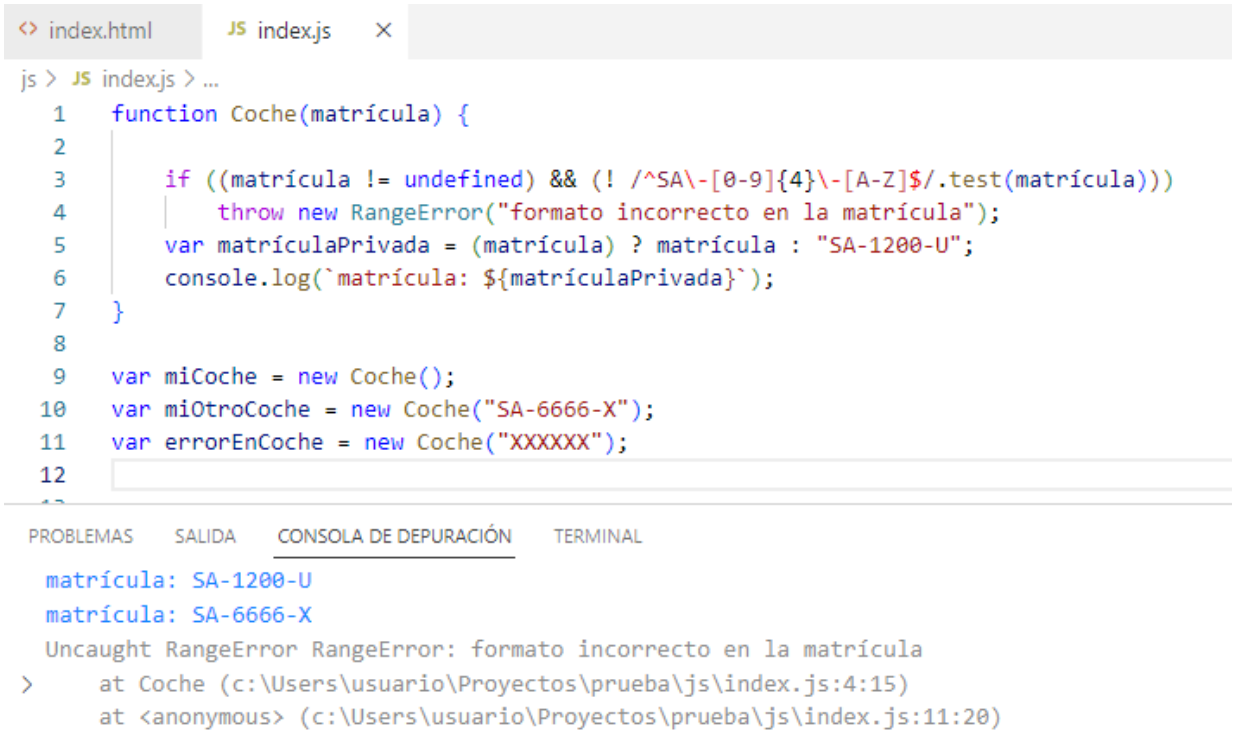
Al hacer **new Coche()** veríamos el mensaje en la consola:



El método constructor puede recibir argumentos que servirán normalmente para inicializar y personalizar el estado de la instancia:

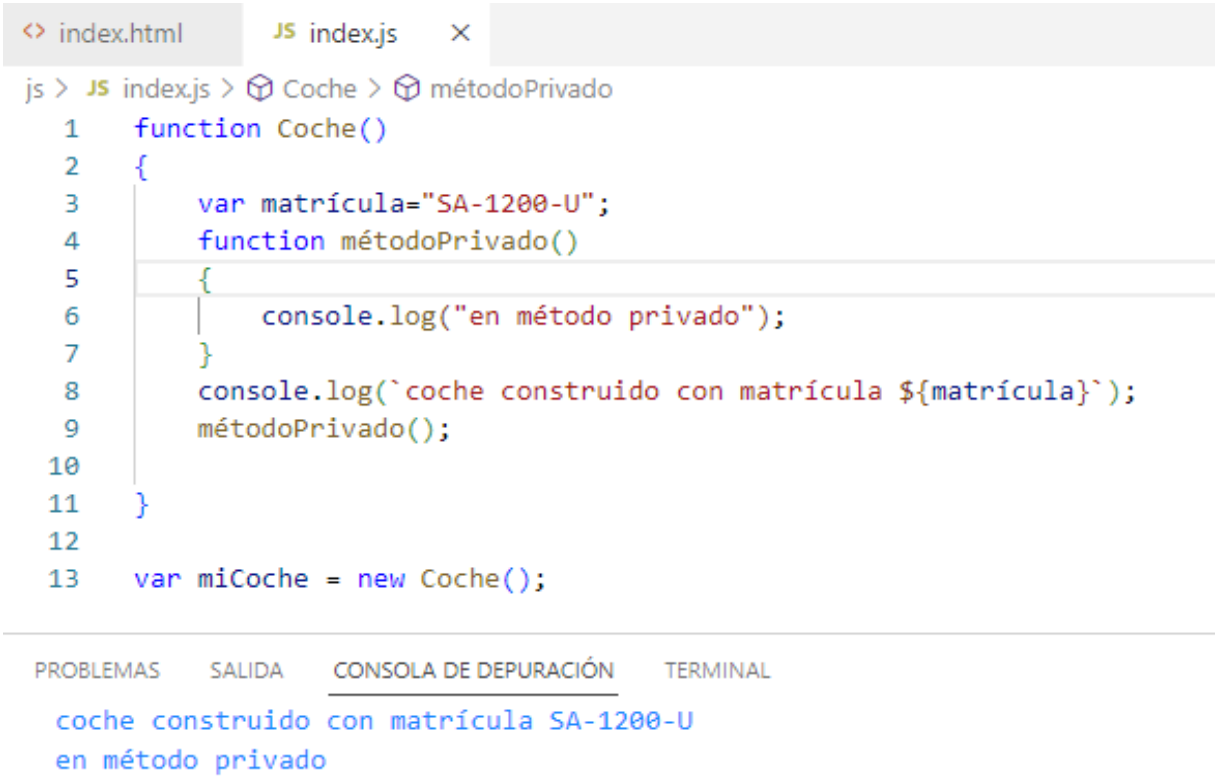
```
function Coche(matrícula) {
```

```
    if ((matrícula != undefined) && (! /^SA\-[0-9]{4}\-[A-Z]$/.test(matrícula)))
       throw new RangeError("formato incorrecto en la matrícula");
    var matrículaPrivada = (matrícula) ? matrícula : "SA-1200-U";
    console.log(`matrícula: ${matrículaPrivada}`);
}

var miCoche = new Coche();
var miOtroCoche = new Coche("SA-6666-X");
var errorEnCoche = new Coche("XXXXXX");
```



En el siguiente ejemplo encontramos la declaración un método **privado** que solo puede ser usado dentro de la función declarativa de la clase:

```
function Coche()
{
   var matrícula="SA-1200-U";
   function métodoPrivado()
   {
      console.log("en método privado");
   }
   console.log(`coche construido con matrícula ${matrícula}`);
   métodoPrivado();

}

var miCoche = new Coche();
```



Si se quiere declarar una propiedad pública de la clase se debe hacer a través de una referencia a la propia instancia haciendo uso de la palabra reservada **this:**

```
function Coche()
{
   this.matrícula="SA-1200-U";
}

var miCoche = new Coche();
var miOtroCoche = new Coche();
miOtroCoche.matrícula="SA-1500-H";
console.log(miCoche.matrícula);
console.log(miOtroCoche.matrícula);
```

index.html index.js

js > index.js > ...

```
1   function Coche()
2   {
3       this.matrícula="SA-1200-U";
4   }
5
6   var miCoche = new Coche();
7   var miOtroCoche = new Coche();
8   miOtroCoche.matrícula="SA-1500-H";
9   console.log(miCoche.matrícula);
10  console.log(miOtroCoche.matrícula);
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

```
SA-1200-U
SA-1500-H
```

Lo mismo para un método público:

index.html index.js

js > index.js > Coche > acelerar

```
1   function Coche()
2   {
3       this.matrícula="SA-1200-U";
4       this.acelerar = function () {
5           console.log("BRRRRRR.......");
6       }
7   }
8
9   var miCoche = new Coche();
10  miCoche.acelerar();
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

```
BRRRRRR.......
```

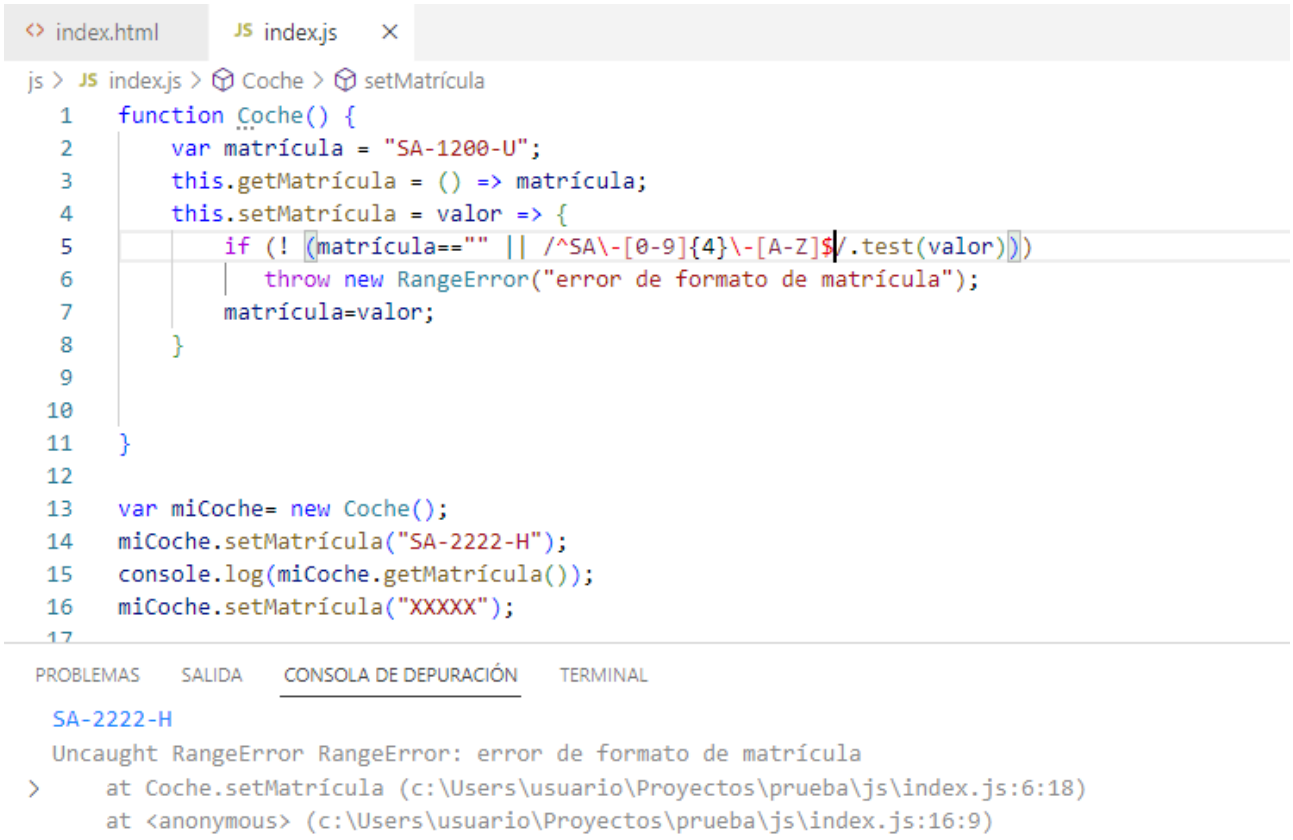
Los métodos de una clase **no se deben declarar** mediante funciones **arrow**. Ya se vio que las funciones **arrow** no disponían de la colección **arguments**, pero además tienen más diferencias. Las dos restantes tienen que ver con la definición de métodos en clases. Dentro de un método de una clase definido mediante **function** la referencia a **this** es a la instancia de la misma, pero en una función **arrow** esta referencia se pierde y apunta al elemento superior en el entorno en el que se utiliza, generalmente apunta a **window**. La otra diferencia es que la utilización de la palabra reservada **super** dentro de un método declarado con **function** apunta a la clase padre de la clase en que se referencia, mientras que en un método definido mediante una función **arrow** esta referencia se pierde. Por lo demás, si no se hace uso de alguno de estos elementos es totalmente válido el uso de funciones **arrow** para definir métodos.

Las propiedades y métodos estáticos se definen asignado directamente la propiedad o método a la clase:

```
Coche.secreto="XF4";
Coche.saluda = () => {
      console.log(`mi secreto es ${Coche.secreto}`);
};

Coche.saluda();
```

Los lenguajes que implementan completamente el paradigma de programación orientada a objetos disponen de lo que se llaman **getters** y **setters** que son "métodos" que son invocados de forma automática cuando se quiere recuperar o modificar, respectivamente, el contenido de una propiedad pública. La forma de implementar **getter** y **setter** en javaScript fue codificar métodos públicos con nombre **get**Propiedad y **set**Propiedad para acceder a la propiedad **privada** con el mismo nombre. Por ejemplo, para la propiedad **matrícula** del caso anterior:

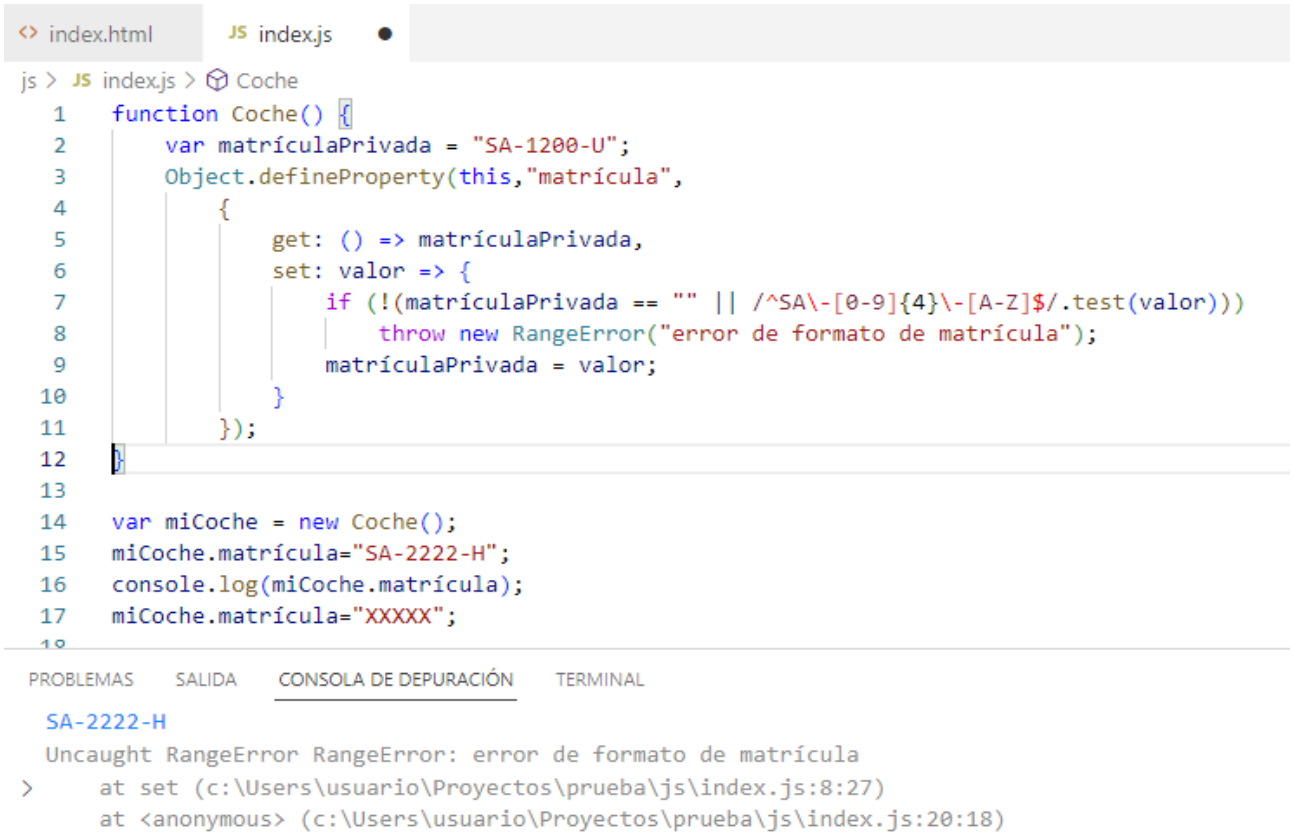
```
function Coche() {
   var matrícula = "SA-1200-U";  // propiedad privada
   this.getMatrícula = () => matrícula;
   this.setMatrícula = valor => {
      if (! (matrícula=="" || /^SA\-[0-9]{4}\-[A-Z]$/.test(valor)))
        throw new RangeError("error de formato de matrícula");
      matrícula=valor;
   }
}
```



Mediante estos métodos se tiene control sobre lo que contienen las propiedades.

También se pueden crear **getter** y **setter** haciendo uso del método **defineProperty** de la clase **Object** del que son herederas todas las clase javaScript, es el antecesor común a todas.

```
function Coche() {
   var matrículaPrivada = "SA-1200-U";
   Object.defineProperty(this,"matrícula",
      {
         get: () => matrículaPrivada,
         set: valor => {
            if (!(matrículaPrivada == "" || /^SA\-[0-9]{4}\-[A-Z]$/.test(valor)))
               throw new RangeError("error de formato de matrícula");
            matrículaPrivada = valor;
         }
      });
}
```



Actualmente ya está implementada la definición de clases mediante la sentencia declarativa **class** y la inclusión de definición de variables y métodos privados y estáticos así como la declaración de **getter** y **setter**.

La clase **Coche,** una propiedad pública **matrícula** con **getter** y **setter,** una propiedad privada **matrículaPrivada** para soporte de la anterior, un método público **acelerar**, un método privado **preCalentar**, una propiedad estática **secreto** sin **getter** ni **setter** y un método estático **saluda** quedaría:

```
class Coche {
  // propiedad estática
  static secreto = "XF4";
  // propiedad privada
  #matrículaPrivada = "SA-1200-U";

  // constructor de la clase
  constructor(númeroMatrícula) {

    if ((númeroMatrícula != undefined) && (! /^SA\-[0-9]{4}\-[A-Z]$/.test(númeroMatrícula)))
      throw new RangeError("formato incorrecto en la matrícula");
    this.#matrículaPrivada = (númeroMatrícula) ? númeroMatrícula : "SA-1200-U";
  }

  // getter y setter para la propiedad pública matrícula
  get matrícula() {
    return this.#matrículaPrivada;
  }
```

```
    set matrícula(valor) {
        if (!(valor == "" || /^SA\-[0-9]{4}\-[A-Z]$/.test(valor)))
            throw new RangeError("error de formato de matrícula");
        this.#matrículaPrivada = valor;
    }

    // método público
    acelerar() {
        this.#precalentar();
        console.log("BRRRR .....");
    }

    // método privado
    #precalentar() {
        console.log("calentando ...");
    }

    // método estático
    static saluda() {
        console.log(`mi secreto es ${Coche.secreto}`)
    }
}

var miCoche = new Coche();

// referencia a propiedad pública
miCoche.matrícula = "SA-6666-K";
console.log(miCoche.matrícula);

var miOtroCoche = new Coche("SA-5555-S");
console.log(miOtroCoche.matrícula);

// llamada método público
miCoche.acelerar();

// llamada propiedad y método estático
Coche.secreto = "XXX";
Coche.saluda();
```

La definición de la clase va encerrada entre llaves tras una declaración **class** *nombreClase*. El constructor de la clase se codifica como una función con nombre **constructor** pudiendo llevar argumentos. Las variables y métodos **privados** deben llevar su nombre precedido de un carácter **#**. Las variable y métodos **estáticos** deben en llevar en su declaración la palabra **static** precediéndola. Un **getter** se codifica como función con el nombre de la propiedad precedido de la palabra **get.** Un **setter** se codifica como función con el nombre de la propiedad precedido de la palabra **set.** Todas las demás declaraciones de propiedades o métodos se considerarán públicos.

Se puede implementar herencia simple en javaScript añadiendo tras la palabra **class** y el nombre de la clase, la palabra **extends** y el nombre de la clase desde la que se hereda, es decir, el nombre de la clase padre. Para ampliar conocimientos sobre declaración y uso de clases se puede consultar https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Classes.

4 - Formato IEEE754

Vamos a hacer una página que sirva de conversor entre números escritos en base decimal y formato **IEEE754** de precisión simple y doble para ver la forma que estos datos quedan almacenados en la memoria y viceversa. Esta página se puede ver en funcionamiento en https://www.manjarr.es/ieee754 en donde también encontraremos una descripción de como se hace esta conversión.

Para encapsular todo el proceso de conversión se han definido dos clases:**IEEE754_Single** y **IEEE754_Double.** Ambas tienen como propiedades públicas: el **número** en base decimal almacenado, el **signo** en binario del número codificado en **IEEE754**, el **exponente** de la codificación del número en formato **IEEE754** y la **mantisa** del mismo. Los requisitos de tamaño y contenidos de estas vienen derivados de la especificación **IEEE754** para ambos tipos de datos, precisión simple o doble.

Se han añadido por conveniencia dos métodos públicos: **toJSON** que devuelve una cadena con el código **JSON** del contenido de la instancia. **JSON** es un código para describir las propiedades y contenidos de estas de un objeto, que se corresponde con la descripción de objetos anónimos de javaScript. Por ejemplo, el código JSON para describir una instancia de objeto con una propiedad **edad** y valor **18** y una propiedad **nombre** y valor **Ángel Sánchez** sería

```
{
  edad: 12,
  nombre: "Ángel Sánchez"
}
```

El otro método que se ha añadido es **toString**. Este método ya existía por venir heredado de **Object**, antecesor de todos los objetos de javaScript, pero no está implementado puesto que no hay un antecesor entre **Object** y el propio objeto que tenga implementado el método **toString.** El método **toString** se invoca de forma automática cuando se precisa una conversión de tipo de datos a cadena. Por ejemplo, si se hace un **alert** de la variable que contiene la instancia, se precisa su conversión a cadena. Es en este momento cuando se invoca de forma automática el método **toString**.

El código para la definición de la clase **IEEE754_Single** podría quedar:

```
class IEEE754_Single {
  // el constructor puede recibir o un Number
  // o tres argumentos de tipo string con representaciones binarias del signo, exponente y mantisa

  // propiedades privadas
  #signo = "0"; // signo del valor almacenado en IEEE754 Single
  #exponente = "0".padEnd(8);    // exponente del valor almacenado en IEEE754 Single
  #mantisa = "0".padEnd(23); // mantisa del valor almacenado en IEEE754 Single

  constructor() {
    switch (arguments.length) {
      case 0:
        break;  // valor inicial 0
      case 1:
        // puede ser un Number
        this.numero = arguments[0];
```

```
            break;
        case 3:
            // tres argumentos de tipo cadena en binario
            this.signo = arguments[0];
            this.exponente = arguments[1];
            this.mantisa = arguments[2];
            break;
        default:
            throw new SyntaxError("incorrecto número de argumentos en el constructor");
    }
}

get signo() {
    return this.#signo;
}
set signo(valor) {
    if (valor == "")
        valor = 0;
    if (! /^[01]$/.test(valor)) {
        throw new TypeError("solo puede ser 0 o 1");
    }
    this.#signo = valor.toString();
}

get exponente() {
    return this.#exponente;
}
set exponente(valor) {
    if (valor == "")
        valor = 0;
    if (! /^[01]{1,8}$/.test(valor)) {
        throw new TypeError("solo puede ser una cadena de entre 0 y 8 ceros y/o unos");
    }
    this.#exponente = (valor.toString()).padStart(8, "0");
}

get mantisa() {
    return this.#mantisa;
}
set mantisa(valor) {
    if (valor == "")
        valor = 0;
    if (! /^[01]+$/.test(valor)) {
        throw new TypeError("solo puede ser una cadena de ceros y/o unos");
    }
    this.#mantisa = ((valor.toString()).padEnd(23, "0")).substring(0, 23); // truncado sin redondeo
}

get numero() {
    let auxExponente = parseInt(this.#exponente, 2) - 127;
    let auxEntero = "";
    let auxDecimal = "";
   // control de las excepciones
    if (parseInt(this.#exponente, 2) == 255)
        if (parseInt(this.#mantisa, 2) == 0)
            return (this.#signo == "1") ? -Infinity : Infinity;
        else
            return NaN;
    if (parseInt(this.#exponente, 2) == 0)
```

```
        if (parseInt(this.#mantisa, 2) == 0)
            return 0;
        else {
            // no normalIzado
            return this.#parseBinaryFloat("0.".padEnd(126, "0") + this.#mantisa);
        }
    if (auxExponente >= 0) {
        auxEntero = ("1" + this.#mantisa).padEnd(auxExponente + 1, "0").substring(0, auxExponente + 1);
        auxDecimal = this.#mantisa.substring(auxExponente);
    }
    else {
        auxEntero = "0";
        auxDecimal = "1".padStart(Math.abs(auxExponente), "0") + this.#mantisa;
    }
    let aux = auxEntero + ((auxDecimal.length > 0) ? "." + auxDecimal : "");

    return this.#parseBinaryFloat(((this.#signo == 1) ? "-" : "") + aux);
}

set numero(valor) {
     // control de las excepciones
    if (isNaN(valor)) {
        this.signo = "0";
        this.exponente = "1".padEnd(8, "1");
        this.mantisa = "1";
        return;
    }
    if (valor == Infinity || valor == -Infinity) {
        this.signo = (valor == Infinity) ? 0 : 1;
        this.exponente = "1".padEnd(8, "1");
        this.mantisa = "0";
        return;
    }
    if (valor == 0) {
        this.signo = "0";
        this.exponente = "0".padEnd(8, "0");
        this.mantisa = "0".padEnd(23, "0");
        return;
    }
    let valorAnterior = {  // salvaguardamos el valor anterior para restaurar si error
        signo: this.signo,
        mantisa: this.mantisa,
        exponente: this.exponente,
        numero: this.numero
    };
    this.signo = ((valor.toString())[0] == "-") ? 1 : 0;
    let valorSinSigno = Number(valor.toString().replace(/^[\+\-]/, ""));
    let valorBinarioSinSigno = valorSinSigno.toString(2);
    let valorBinarioSinSignoSinPunto = valorBinarioSinSigno.replace(".", "");
    let posiciónPunto = valorBinarioSinSigno.indexOf(".");
    let posiciónPrimer1 = valorBinarioSinSignoSinPunto.indexOf("1");
    this.mantisa = (posiciónPrimer1 == -1) ? "0" :
                                    valorBinarioSinSignoSinPunto.substring(posiciónPrimer1 + 1);
    try {
        if (posiciónPunto == -1) {
            if (valorBinarioSinSignoSinPunto.length > 128) {
                this.exponente = "1".padEnd(8, "1");
                this.mantisa = "0";
                return;
```

```
                }
                this.exponente = (127 + (valorBinarioSinSignoSinPunto.length - 1)).toString(2);
            }
            else {
                this.exponente = (-(posiciónPrimer1 - posiciónPunto + 1) + 127).toString(2);
            }
        } catch (error) {
            this.signo = valorAnterior.signo;
            this.mantisa = valorAnterior.mantisa;
            this.exponente = valorAnterior.exponente;
            throw new RangeError("valor excede el rango admitido");
        }
    }

    // método privado que devuelve el valor decimal de un número en binario
    #parseBinaryFloat(valor) {
        if (! /^[\+\-]?[01]+(\.[01]+)?$/.test(valor)) {
            throw new TypeError("solo puede ser una cadena ceros y/o unos y un punto decimal opcional");
        }
        var res = valor.split('.');
        return res.length < 2 ?
            parseInt(valor, 2) :
            parseInt(res[0] + res[1], 2) / Math.pow(2, res[1].length);
    }

    toJSON() {
        return `{
    signo: "${this.signo}",
    exponente: "${this.exponente}",
    mantisa: "${this.mantisa}",
    numero: "${this.numero}"
}`;
    }

    toString() {
        return `${this.signo}:${this.exponente}:${this.mantisa}:${this.numero}`;
    }
}
```

El **constructor** de la clase puede recibir **0, 1** u **3** argumentos. Si no recibe argumento se inicializa el valor almacenado a **0**. Si **1** se espera que sea un **number** con un valor decimal, y si **3** se espera que sean tres cadenas representando la primera el signo, la segunda el exponente y la tercera la mantisa, todas en binario correspondiente a la codificación **IEEE754** del número almacenado.

Aparecen los **getter** y **setter** de las cuatro propiedades públicas para controlar el contenido exigiendo que se ajusten a la especificación **IEEE754**. Nótese que la propiedad **número** no está soportada por una propiedad privada, como las otras, sino que es calculada a partir de los datos de las otras 3.

El método privado **#parseBinaryFloat** se encarga de convertir a número decimal con parte fraccionaria una codificación en binario con parte fraccionaria. La función **parseInt** si es capaz de tomar un código binario y transformarlo a número, pero entero. Esta capacidad no la tiene la función **parseFloat** de ahí la necesidad de este método privado.

Se utiliza varias veces el método **padEnd** de la clase **String** que rellena una cadena por el final con el carácter o cadena que se indique hasta alcanzar la longitud que se indique también.

El código completo de la página podría quedar:

ieee754.html

```
<!DOCTYPE html>
<html lang="es">

<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>IEEE754</title>
   <link rel="preconnect" href="https://fonts.googleapis.com">
   <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
   <link href="https://fonts.googleapis.com/css2?family=Caveat:wght@500&display=swap"
rel="stylesheet">
   <link rel="stylesheet" href="css/ieee754.css">
</head>

<body>
   <main>
      <header>
         <h1>Formato IEEE754</h1>
         <div id="divFecha"></div>
      </header>
      <section>
         <h1>número</h1>
         <div class="número">
            <input type="text" id="txtNúmero">
            <button id="btnNúmero" title="convertir">&#128260;</button>
         </div>
      </section>
      <section>
         <h1>IEEE754 precisión simple (32bits)</h1>
         <h2 class="labelIEEESimple">
            <div class="labelSigno">S</div>
            <div class="labelExponente">exponente</div>
            <div class="labelMantisa">mantisa</div>
         </h2>
         <div class="ieeeSingle">
            <div class="signo">
               <input type="text" maxlength="1">
            </div>
            <div class="exponente">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
            </div>
            <div class="mantisa">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
               <input type="text" maxlength="1">
```

```
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
        </div>
        <button id="btnSingle" title="convertir">&#128260;</button>
</section>
<section>
    <h1>IEEE754 precisión doble (32bits)</h1>
    <h2 class="labelIEEEDoble">
        <div class="labelSigno">S</div>
        <div class="labelExponente">exponente</div>
        <div class="labelMantisa">mantisa</div>
    </h2>
    <div class="ieeeDouble">
        <div class="signo">
            <input type="text" maxlength="1">
        </div>
        <div class="exponente">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
        </div>
        <div class="mantisa">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
            <input type="text" maxlength="1">
```

```
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
                <input type="text" maxlength="1">
            </div>
            <button id="btnDouble" title="convertir">&#128260;</button>
        </div>
    </section>
    <section>
        <div id="divMensaje"></div>
    </section>
  </main>
  <script src="js/mathTools.js"></script>
  <script src="js/ieee754.js"></script>
</body>
```

ieee754.css

```
@charset "UTF-8";
@font-face {
  font-family: "Calibri";
  font-style: normal;
  font-weight: 300;
  font-display: swap;
```

```
  src: url("../webfonts/calibriLi.ttf") format("truetype");
}
* {
  font-family: Calibri, sans-serif;
  font-size: 12pt;
  margin: 0px;
  padding: 0px;
  box-sizing: border-box;
}

body {
  width: 1150px;
  margin: 15px;
}
body > main > header > h1 {
  font-size: 3em;
  margin-bottom: 20px;
  color: blue;
  font-family: "Caveat", cursive;
}
body > main > header > div#divFecha {
  font-size: 9pt;
  text-align: right;
  padding-right: 10px;
  margin-bottom: 20px;
}
body > main > section > h1 {
  font-size: 1.3em;
  color: darkgrey;
  font-weight: bold;
  margin-bottom: 10px;
}
body > main > section > h2.labelIEEEDoble > div.labelMantisa {
  margin-left: 50px;
}
body > main > section > h2 > div {
  font-size: 10pt;
  font-weight: bold;
  display: inline-block;
}
body > main > section > h2 > div.labelExponente {
  width: 138px;
  margin-left: 15px;
}
body > main > section > div {
  margin-bottom: 20px;
}
body > main > section > div.número > input[type=text] {
  display: inline-block;
  width: 320px;
  padding: 3px;
  margin-right: 5px;
}
body > main > section > div.número > button {
  display: inline-block;
  font-size: 25pt;
  margin-left: 15px;
  width: 20px;
  border: none;
```

```
  cursor: pointer;
}
body > main > section > div.ieeeSingle > div, body > main > section > div.ieeeDouble > div {
  font-size: 9pt;
  display: inline-block;
  margin-left: 10px;
}
body > main > section > div.ieeeSingle > div.signo, body > main > section > div.ieeeDouble > div.signo {
  margin-left: 0px;
}
body > main > section > div.ieeeSingle > div > input[type=text], body > main > section > div.ieeeDouble >
div > input[type=text] {
  display: inline-block;
  width: 13px;
  padding: 1px;
  border-spacing: 1px;
  border-width: 1px;
  border-radius: 3px;
}
body > main > section > div.ieeeSingle > button, body > main > section > div.ieeeDouble > button {
  display: inline-block;
  font-size: 25pt;
  margin-left: 15px;
  width: 20px;
  border: none;
  cursor: pointer;
}
body > main > section > div#divMensaje {
  color: red;
  font-size: 11pt;
  height: 50px;
}
body > main > article > h1 {
  font-size: 1.5em;
  color: darkgrey;
  margin-top: 50px;
  margin-bottom: 10px;
}
body > main > article > h2 {
  font-size: 1.25em;
  color: darkgrey;
  margin-top: 20px;
  margin-bottom: 10px;
}
body > main > article > p {
  margin-bottom: 10px;
}
body > main > article > p sup {
  font-size: 8pt;
}
body > main > article > p.centrado {
  text-align: center;
}
body > main > article > p > img {
  margin-top: 15px;
  width: 75%;
}
body > main > article > ul {
  margin-left: 30px;
```

```
  margin-bottom: 20px;
}
body > main > article > ul > li {
  padding-bottom: 4px;
}
body > main > article > pre {
  margin: 20px;
  font-size: 10pt;
  font-weight: bold;
}
body > main > article > pre > sup {
  font-size: 8pt;
}
```

ieee754.js

```
// fecha actual
document.getElementById("divFecha").innerHTML =
    (new Date()).toLocaleDateString("es-ES",
        { weekday: "long", day: "numeric", month: "long", year: "numeric" });

// referencias a elementos
const txtNumero = document.getElementById("txtNúmero");
const btnNúmero = document.getElementById("btnNúmero");
const divMensaje = document.getElementById("divMensaje");
const divIEEESingle = document.querySelector("body>main>section>div.ieeeSingle");
const divIEEEDouble = document.querySelector("body>main>section>div.ieeeDouble");

const divSignoSingle = document.querySelector("body>main>section>div.ieeeSingle>div.signo>input");
const aDivExponenteSingle =
document.querySelectorAll("body>main>section>div.ieeeSingle>div.exponente>input");
const aDivMantisaSingle =
document.querySelectorAll("body>main>section>div.ieeeSingle>div.mantisa>input");
const btnSingle = document.getElementById("btnSingle");

const divSignoDouble =
document.querySelector("body>main>section>div.ieeeDouble>div.signo>input");
const aDivExponenteDouble =
document.querySelectorAll("body>main>section>div.ieeeDouble>div.exponente>input");
const aDivMantisaDouble =
document.querySelectorAll("body>main>section>div.ieeeDouble>div.mantisa>input");
const btnDouble = document.getElementById("btnDouble");

// función para solo aceptar en los keypress datos binarios
const soloBinario = (e) => {
    if (e.target.nodeName == "INPUT") {
        divMensaje.innerText = "";
        if (!(e.key == "0" || e.key == "1"))
            e.preventDefault();
    }
}

// para hacer que cuando coja el foco una caja de texto quede
// seleccionado su contenido
const seleccionado = (e) => {
    if (e.target.nodeName == "INPUT")
        e.target.select();
};
```

```
// limpiar las cajas de texto de Single
const limpiarSingle = () => {
   divSignoSingle.value = "";
   for (let div of aDivExponenteSingle)
      div.value = "";
   for (let div of aDivMantisaSingle)
      div.value = "";
}

// limpiar las cajas de texto de Double
const limpiarDouble = () => {
   divSignoDouble.value = "";
   for (let div of aDivExponenteDouble)
      div.value = "";
   for (let div of aDivMantisaDouble)
      div.value = "";
}

// limpiar la caja de texto de Número
const limpiarNúmero = () => {
   txtNúmero.value = "";
}

// solo admitir caracteres compatibles con un número en decimal
txtNúmero.addEventListener("keypress",
   (e) => {
      if (! /[0-9\.\,eE\+\-]/.test(e.key))
         e.preventDefault();
   }
);

// si se modifica el contenido de número
// borrar datos que pudiera haber en Single y Double
txtNúmero.addEventListener("input",
   (e) => {
      // quitar mensaje error
      divMensaje.innerText = "";
      // limpiar bits
      limpiarSingle();
      limpiarDouble();
   }
);

txtNumero.addEventListener("focus", seleccionado);

// si se pulsa , como separador decimal cambiarlo por .
txtNumero.addEventListener("focusout",
   () => {
      txtNúmero.value = txtNúmero.value.replace(",", ".");
   });

// convertir a Single y Double
btnNúmero.onclick = () => {
   if (txtNúmero.value == "")
      txtNumero.value = "0";
   if (isNaN(txtNúmero.value)) {
      divMensaje.innerText = "formato de número incorrecto";
      txtNúmero.select();
   }
```

```
    else {
        const bitsSingle = new IEEE754_Single(parseFloat(txtNúmero.value));
        const bitsDouble = new IEEE754_Double(parseFloat(txtNúmero.value))
        if (bitsSingle.numero == Infinity || bitsSingle.numero == -Infinity) {
            divMensaje.innerText = "valor Infinity en precisión simple";
            txtNúmero.value = Infinity;
        }
        // poner bits
        divSignoSingle.value = bitsSingle.signo

        for (let i = 0; i < aDivExponenteSingle.length; i++)
            aDivExponenteSingle[i].value = bitsSingle.exponente[i];


        for (let i = 0; i < aDivMantisaSingle.length; i++)
            aDivMantisaSingle[i].value = bitsSingle.mantisa[i];

        if (bitsDouble.numero == Infinity || bitsDouble.numero == -Infinity) {
            divMensaje.innerHTML += "<br>valor Infinity en precisión doble";
            txtNúmero.value = Infinity;
        }
        //poner bits
        divSignoDouble.value = bitsDouble.signo;

        for (let i = 0; i < aDivExponenteDouble.length; i++)
            aDivExponenteDouble[i].value = bitsDouble.exponente[i];

        for (let i = 0; i < aDivMantisaDouble.length; i++)
            aDivMantisaDouble[i].value = bitsDouble.mantisa[i];
    }
}

// convertir a Double y a número
btnSingle.addEventListener("click", () => {
    let bitsSingleExponente = "";
    for (input of aDivExponenteSingle)
        bitsSingleExponente += input.value;
    let bitsSingleMantisa = "";
    for (input of aDivMantisaSingle)
        bitsSingleMantisa += input.value;
    const bitsSingle = new IEEE754_Single(divSignoSingle.value, bitsSingleExponente,
bitsSingleMantisa);
    txtNúmero.value = bitsSingle.numero;
    const bitsDouble = new IEEE754_Double(parseFloat(txtNúmero.value));
    // poner bits
    // se ponen los de single por si se han dejado en balnco algún bit
    divSignoSingle.value = bitsSingle.signo

    for (let i = 0; i < aDivExponenteSingle.length; i++)
        aDivExponenteSingle[i].value = bitsSingle.exponente[i];

    for (let i = 0; i < aDivMantisaSingle.length; i++)
        aDivMantisaSingle[i].value = bitsSingle.mantisa[i];

    // mostrar bits double
    divSignoDouble.value = bitsDouble.signo;

    for (let i = 0; i < aDivExponenteDouble.length; i++)
        aDivExponenteDouble[i].value = bitsDouble.exponente[i];
```

```
    for (let i = 0; i < aDivMantisaDouble.length; i++)
        aDivMantisaDouble[i].value = bitsDouble.mantisa[i];
});

// convertir a Single y a número
btnDouble.addEventListener("click", () => {
    let bitsDoubleExponente = "";
    for (input of aDivExponenteDouble)
        bitsDoubleExponente += input.value;
    let bitsDoubleMantisa = "";
    for (input of aDivMantisaDouble)
        bitsDoubleMantisa += input.value;
    const bitsDouble = new IEEE754_Double(divSignoDouble.value, bitsDoubleExponente,
bitsDoubleMantisa);
    txtNúmero.value = bitsDouble.numero;
    const bitsSingle = new IEEE754_Single(parseFloat(txtNúmero.value));

    if (bitsSingle.numero == Infinity || bitsSingle.numero == -Infinity)
        divMensaje.innerText = "valor Infinity en precisión simple";
    if (bitsDouble.numero == Infinity || bitsDouble.numero == -Infinity)
        divMensaje.innerHTML += "<br>valor Infinity en precisión doble";

    // poner bits
    divSignoSingle.value = bitsSingle.signo
    for (let i = 0; i < aDivExponenteSingle.length; i++)
        aDivExponenteSingle[i].value = bitsSingle.exponente[i];
    for (let i = 0; i < aDivMantisaSingle.length; i++)
        aDivMantisaSingle[i].value = bitsSingle.mantisa[i];

    // mostrar bits double
    // se ponen los de double por si se han dejado en blanco algún bit
    divSignoDouble.value = bitsDouble.signo;
    for (let i = 0; i < aDivExponenteDouble.length; i++)
        aDivExponenteDouble[i].value = bitsDouble.exponente[i];
    for (let i = 0; i < aDivMantisaDouble.length; i++)
        aDivMantisaDouble[i].value = bitsDouble.mantisa[i];

});

// permitir solo 0 y 1
divIEEESingle.addEventListener("keypress", soloBinario);

divIEEESingle.addEventListener("focus", seleccionado, true);

// cualquier cambio en Single borra Double y número
divIEEESingle.addEventListener("input", () => {
    // quitar mensaje error
    divMensaje.innerText = "";
    // limpiar número y double
    limpiarNúmero();
    limpiarDouble();
}, true);

// permitir solo 0 y 1
divIEEEDouble.addEventListener("keypress", soloBinario);

divIEEEDouble.addEventListener("focus", seleccionado, true);
```

```
// cualquier cambio en Double borra Single y número
divIEEEDouble.addEventListener("input", () => {
   // quitar mensaje error
   divMensaje.innerText = "";
   // limpiar número y double
   limpiarNúmero();
   limpiarSingle();
}, true);
```

mathTools.js

```
class IEEE754_Single {
   // el constructor puede recibir o un Number
   // o tres argumentos de tipo string con representaciones binarias del signo, exponente y mantisa

   // propiedades privadas
   #signo = "0"; // signo del valor almacenado en IEEE754 Single
   #exponente = "0".padEnd(8);    // exponente del valor almacenado en IEEE754 Single
   #mantisa = "0".padEnd(23); // mantisa del valor almacenado en IEEE754 Single

   constructor() {
      switch (arguments.length) {
         case 0:
            break;  // valor inicial 0
         case 1:
            // puede ser un Number
            this.numero = arguments[0];
            break;
         case 3:
            // tres argumentos de tipo cadena en binario
            this.signo = arguments[0];
            this.exponente = arguments[1];
            this.mantisa = arguments[2];
            break;
         default:
            throw new SyntaxError("incorrecto número de argumentos en el constructor");
      }
   }

   get signo() {
      return this.#signo;
   }
   set signo(valor) {
      if (valor == "")
         valor = 0;
      if (! /^[01]$/.test(valor)) {
         throw new TypeError("solo puede ser 0 o 1");
      }
      this.#signo = valor.toString();
   }

   get exponente() {
      return this.#exponente;
   }
   set exponente(valor) {
      if (valor == "")
         valor = 0;
      if (! /^[01]{1,8}$/.test(valor)) {
         throw new TypeError("solo puede ser una cadena de entre 0 y 8 ceros y/o unos");
```

```
    }
    this.#exponente = (valor.toString()).padStart(8, "0");
}

get mantisa() {
    return this.#mantisa;
}
set mantisa(valor) {
    if (valor == "")
        valor = 0;
    if (! /^[01]+$/.test(valor)) {
        throw new TypeError("solo puede ser una cadena de ceros y/o unos");
    }
    this.#mantisa = ((valor.toString()).padEnd(23, "0")).substring(0, 23); // truncado sin redondeo
}

get numero() {
    let auxExponente = parseInt(this.#exponente, 2) - 127;
    let auxEntero = "";
    let auxDecimal = "";
    // control de las excepciones
    if (parseInt(this.#exponente, 2) == 255)
        if (parseInt(this.#mantisa, 2) == 0)
            return (this.#signo == "1") ? -Infinity : Infinity;
        else
            return NaN;
    if (parseInt(this.#exponente, 2) == 0)
        if (parseInt(this.#mantisa, 2) == 0)
            return 0;
        else {
            // no normalIzado
            return this.#parseBinaryFloat("0.".padEnd(126, "0") + this.#mantisa);
        }
    if (auxExponente >= 0) {
        auxEntero = ("1" + this.#mantisa).padEnd(auxExponente + 1, "0").substring(0, auxExponente + 1);
        auxDecimal = this.#mantisa.substring(auxExponente);
    }
    else {
        auxEntero = "0";
        auxDecimal = "1".padStart(Math.abs(auxExponente), "0") + this.#mantisa;
    }
    let aux = auxEntero + ((auxDecimal.length > 0) ? "." + auxDecimal : "");

    return this.#parseBinaryFloat(((this.#signo == 1) ? "-" : "") + aux);
}

set numero(valor) {
    // control de las excepciones
    if (isNaN(valor)) {
        this.signo = "0";
        this.exponente = "1".padEnd(8, "1");
        this.mantisa = "1";
        return;
    }
    if (valor == Infinity || valor == -Infinity) {
        this.signo = (valor == Infinity) ? 0 : 1;
        this.exponente = "1".padEnd(8, "1");
        this.mantisa = "0";
        return;
```

```
        }
        if (valor == 0) {
            this.signo = "0";
            this.exponente = "0".padEnd(8, "0");
            this.mantisa = "0".padEnd(23, "0");
            return;
        }
        let valorAnterior = {  // salvaguardamos el valor anterior para restaurar si error
            signo: this.signo,
            mantisa: this.mantisa,
            exponente: this.exponente,
            numero: this.numero
        };
        this.signo = ((valor.toString())[0] == "-") ? 1 : 0;
        let valorSinSigno = Number(valor.toString().replace(/^[\+\-]/, ""));
        let valorBinarioSinSigno = valorSinSigno.toString(2);
        let valorBinarioSinSignoSinPunto = valorBinarioSinSigno.replace(".", "");
        let posiciónPunto = valorBinarioSinSigno.indexOf(".");
        let posiciónPrimer1 = valorBinarioSinSignoSinPunto.indexOf("1");
        this.mantisa = (posiciónPrimer1 == -1) ? "0" :
valorBinarioSinSignoSinPunto.substring(posiciónPrimer1 + 1);
        try {
            if (posiciónPunto == -1) {
                if (valorBinarioSinSignoSinPunto.length > 128) {
                    this.exponente = "1".padEnd(8, "1");
                    this.mantisa = "0";
                    return;

                }
                this.exponente = (127 + (valorBinarioSinSignoSinPunto.length - 1)).toString(2);
            }
            else {
                this.exponente = (-(posiciónPrimer1 - posiciónPunto + 1) + 127).toString(2);
            }
        } catch (error) {
            this.signo = valorAnterior.signo;
            this.mantisa = valorAnterior.mantisa;
            this.exponente = valorAnterior.exponente;
            throw new RangeError("valor excede el rango admitido");
        }
    }

    // método privado que devuelve el valor decimal de un número en binario
    #parseBinaryFloat(valor) {
        if (! /^[\+\-]?[01]+(\.[01]+)?$/.test(valor)) {
            throw new TypeError("solo puede ser una cadena ceros y/o unos y un punto decimal opcional");
        }
        var res = valor.split('.');
        return res.length < 2 ?
            parseInt(valor, 2) :
            parseInt(res[0] + res[1], 2) / Math.pow(2, res[1].length);
    }

    toJSON() {
        return `{
    signo: "${this.signo}",
    exponente: "${this.exponente}",
    mantisa: "${this.mantisa}",
    numero: "${this.numero}"
```

```
}`;
    }

    toString() {
        return `${this.signo}:${this.exponente}:${this.mantisa}:${this.numero}`;
    }
}

class IEEE754_Double {
    // el constructor puede recibir o un Number
    // o tres argumentos de tipo string con representaciones binarias del signo, exponente y mantisa

    // propiedades privadas
    #signo = "0"; // signo del valor almacenado en IEEE754 Single
    #exponente = "0".padEnd(11);    // exponente del valor almacenado en IEEE754 Single
    #mantisa = "0".padEnd(52); // mantisa del valor almacenado en IEEE754 Single

    constructor() {
        switch (arguments.length) {
            case 0:
                break;  // valor inicial 0
            case 1:
                // puede ser un Number
                this.numero = arguments[0];
                break;
            case 3:
                // tres argumentos de tipo cadena en binario
                this.signo = arguments[0];
                this.exponente = arguments[1];
                this.mantisa = arguments[2];
                break;
            default:
                throw new SyntaxError("incorrecto número de argumentos en el constructor");
        }
    }

    get signo() {
        return this.#signo;
    }
    set signo(valor) {
        if (valor == "")
            valor = 0;
        if (! /^[01]$/.test(valor)) {
            throw new TypeError("solo puede ser 0 o 1");
        }
        this.#signo = valor.toString();
    }

    get exponente() {
        return this.#exponente;
    }
    set exponente(valor) {
        if (valor == "")
            valor = 0;
        if (! /^[01]{1,11}$/.test(valor)) {
            throw new TypeError("solo puede ser una cadena de entre 0 y 11 ceros y/o unos");
        }
        this.#exponente = (valor.toString()).padStart(11, "0");
    }
```

```
get mantisa() {
   return this.#mantisa;
}
set mantisa(valor) {
   if (valor == "")
      valor = 0;
   if (! /^[01]+$/.test(valor)) {
      throw new TypeError("solo puede ser una cadena de ceros y/o unos");
   }
   this.#mantisa = ((valor.toString()).padEnd(52, "0")).substring(0, 52); // truncado sin redondeo
}

get numero() {
   let auxExponente = parseInt(this.#exponente, 2) - 1023;
   let auxEntero = "";
   let auxDecimal = "";
   // control de las excepciones
   if (parseInt(this.#exponente, 2) == 2047)
      if (parseInt(this.#mantisa, 2) == 0)
         return (this.#signo == "1") ? -Infinity : Infinity;
      else
         return NaN;
   if (parseInt(this.#exponente, 2) == 0)
      if (parseInt(this.#mantisa, 2) == 0)
         return 0;
      else {
         // no normalIzado
         return this.#parseBinaryFloat("0.".padEnd(1022, "0") + this.#mantisa);
      }
   if (auxExponente >= 0) {
      auxEntero = ("1" + this.#mantisa).padEnd(auxExponente + 1, "0").substring(0, auxExponente + 1);
      auxDecimal = this.#mantisa.substring(auxExponente);
   }
   else {
      auxEntero = "0";
      auxDecimal = "1".padStart(Math.abs(auxExponente), "0") + this.#mantisa;
   }
   let aux = auxEntero + ((auxDecimal.length > 0) ? "." + auxDecimal : "");

   return this.#parseBinaryFloat(((this.#signo == 1) ? "-" : "") + aux);
}

set numero(valor) {
   // control de las excepciones
   if (isNaN(valor)) {
      this.signo = "0";
      this.exponente = "1".padEnd(11, "1");
      this.mantisa = "1";
      return;
   }
   if (valor == Infinity || valor == -Infinity) {
      this.signo = (valor == Infinity) ? 0 : 1;
      this.exponente = "1".padEnd(11, "1");
      this.mantisa = "0";
      return;
   }
   if (valor == 0) {
      this.signo = 0;
```

```
            this.exponente = "0".padEnd(11, "0");
            this.mantisa = "0";
            return;
        }
        let valorAnterior = {  // salvaguardamos el valor anterior para restaurar si error
            signo: this.signo,
            mantisa: this.mantisa,
            exponente: this.exponente,
            numero: this.numero
        };
        this.signo = ((valor.toString())[0] == "-") ? 1 : 0;
        let valorSinSigno = Number(valor.toString().replace(/^[\+\-]/, ""));
        let valorBinarioSinSigno = valorSinSigno.toString(2);
        let valorBinarioSinSignoSinPunto = valorBinarioSinSigno.replace(".", "");
        let posiciónPunto = valorBinarioSinSigno.indexOf(".");
        let posiciónPrimer1 = valorBinarioSinSignoSinPunto.indexOf("1");
        this.mantisa = (posiciónPrimer1 == -1) ? "0" :
valorBinarioSinSignoSinPunto.substring(posiciónPrimer1 + 1);
        try {
            if (posiciónPunto == -1) {
                if (valorBinarioSinSignoSinPunto.length > 1024) {
                    this.exponente = "1".padEnd(11, "1");
                    this.mantisa = "0";
                    return;
                }
                this.exponente = (1023 + (valorBinarioSinSignoSinPunto.length - 1)).toString(2);
            }
            else {
                this.exponente = (-(posiciónPrimer1 - posiciónPunto + 1) + 1023).toString(2);
            }
        } catch (error) {
            this.signo = valorAnterior.signo;
            this.mantisa = valorAnterior.mantisa;
            this.exponente = valorAnterior.exponente;
            throw new RangeError("valor excede el rango admitido");
        }
    }

    // método privado que devuelve el valor decimal de un número en binario
    #parseBinaryFloat(valor) {
        if (! /^[\+\-]?[01]+(\.[01]+)?$/.test(valor)) {
            throw new TypeError("solo puede ser una cadena ceros y/o unos y un punto decimal opcional");
        }
        var res = valor.split('.');
        return res.length < 2 ?
            parseInt(valor, 2) :
            parseInt(res[0] + res[1], 2) / Math.pow(2, res[1].length);
    }

    toJSON() {
        return `{
    signo: "${this.signo}",
    exponente: "${this.exponente}",
    mantisa: "${this.mantisa}",
    numero: "${this.numero}"
}`;
    }

    toString() {
```

```
        return `${this.signo}:${this.exponente}:${this.mantisa}:${this.numero}`;
    }
}
```

En **ieee754.js** encontramos el siguiente código para restringir la pulsación en teclado a solo datos en binario en cualquiera de las cajas de texto, tanto de Single como de Double:

```
// función para solo aceptar en los keypress datos binarios
const soloBinario = (e) => {
    if (e.target.nodeName == "INPUT") {
        divMensaje.innerText = "";
        if (!(e.key == "0" || e.key == "1"))
            e.preventDefault();
    }
}

divIEEESingle.addEventListener("keypress", soloBinario);
divIEEEDouble.addEventListener("keypress", soloBinario);
```

Si nos fijamos, la captura del evento **keypress** no ha sido hecha en cada uno de **input** sino en un elemento superior que los contiene. Cuando se desencadena un evento en un elemento de una página se dice que **burbujea** porque ese evento va pasando por todos los elementos antecesores del elemento en el que se desencadena. Cuando se pulsa una tecla en una caja de texto de Single, por ejemplo, el evento primero pasa por **html**, luego por **body,** luego por **section …** así hasta llegar al **input** y luego vuelve a burbujear hasta **hrml** deshaciendo el camino. El evento puede ser capturado en cualquiera de los dos recorridos. Por defecto, cuando se captura con **addEventListener** se captura en la fase de ascenso. Si se desease capturar en la fase de descenso habría que añadir un tercer argumento en el método **addEventListener** con valor **true**. El evento se puede capturar en todos los elementos intermedios. La captura en descenso la podemos ver al capturar el evento **focus** de las cajas de texto para que quede su contenido seleccionado:

```
divIEEESingle.addEventListener("focus", seleccionado, true);
```

¿Por qué se ha capturado aquí en descenso? Porque el evento **focus** en un elemento que no coge el foco como es **DIV** solo puede ser capturado en la fase de descenso, sin embargo el evento **keypress** si que puede ser capturado en la fase de ascenso en el elemento **DIV.**

Si dentro de un **DIV** hay varias cajas de texto u otros elementos, ¿como se sabe en que elemento se desencadenó el evento? En nuestro caso ¿en caja de texto?. Ya sabemos que cuando se desencadena un evento la función de **callback** que se ejecuta, que es donde incluimos el código de respuesta al evento, puede llevar un argumento que contiene información sobre el evento producido. Pues bien este argumento, que es una instancia de la clase **Event**, tiene una propiedad llamada **target** que tiene una referencia al elemento en el que se desencadenó el evento.

Para asegurarnos de que el evento se produjo en un elemento de tipo **INPUT** es por lo que aparece en el código:

```
if (e.target.nodeName == "INPUT")
```

nodeName es una propiedad que tienen todos los elementos|nodos del **DOM** que indican que tipo de tag html es. Siempre aparecen en mayúsculas.

El evento **input** se desencadena en una caja de texto cuando cambia su contenido, al que se accede a través de su propiedad **value,** por cualquier motivo. Hay un evento **change** que se desencadena también al detectar un cambio en el contenido de la caja de texto pero solo cuando la caja de texto pierde el foco. Aquí nos interesa que se desencadena con cualquier cambio por eso se ha optado por el evento **input**.

El método privado **#parseBinaryFloat** vemos que aparece repetido en la declaración de ambas clases. También vimos que cubre una carencia de la función **parseFloat**. Sería interesante disponer de esta función en futuros usos. Como primera idea podríamos codificar la función fuera de la definición de las clases y ya tendríamos acceso a ella sin más que incorporar el archivo js en el que se encuentre declarada.

```
parseBinaryFloat(valor) {
   if (! /^[\+\-]?[01]+(\.[01]+)?$/.test(valor)) {
      throw new TypeError("solo puede ser una cadena ceros y/o unos y un punto decimal opcional");
   }
   var res = valor.split('.');
   return res.length < 2 ?
      parseInt(valor, 2) :
      parseInt(res[0] + res[1], 2) / Math.pow(2, res[1].length);
}
```

A medida que vayamos desarrollando código nos vamos a encontrar con que tenemos una serie de funciones de utilidad desarrolladas por nosotros referidas a un tema particular, por ejemplo, a cálculo estadístico, o cálculo empresarial. En lugar de tener multitud de funciones sin relación aparente lo ideal será encapsularlas bajo una clase común. Esto es lo que hicieron los diseñadores de javaScript con la clase **Math**. Es una clase formada por métodos estáticos que agrupa una serie de funciones relacionadas con el cálculo matemático. Podría parecer interesante añadir nuestra función **parseBinaryFloat** a esta colección de métodos de la clase **Math,** pero la clase **Math** ya está declarada y además no tenemos acceso al código en el que se declara. JavaScript es muy permisivo en cuanto a estas cosas y permite añadir o eliminar métodos y propiedades de objetos de forma dinámica. En nuestro caso, no hay ningún problema en escribir:

```
Math.parseBinaryFloat(valor) {
   if (! /^[\+\-]?[01]+(\.[01]+)?$/.test(valor)) {
      throw new TypeError("solo puede ser una cadena ceros y/o unos y un punto decimal opcional");
   }
   var res = valor.split('.');
   return res.length < 2 ?
      parseInt(valor, 2) :
      parseInt(res[0] + res[1], 2) / Math.pow(2, res[1].length);
}
```

La clase **Math,** a partir de la ejecución de este código, ya dispone del método estático **parseBinaryFloat**. La ejecución del código anterior no consolida la inclusión del método. Si hacemos referencia en otro programa a esta función nos dirá que no existe, a no ser que se vuelva a ejecutar el código anterior. Esto que se ha hecho con **Math** es aplicable a cualquier clase o instancia de objeto, se estará creando un método estático si se refiere a una clase y a un método particular si se refiere a una instancia.

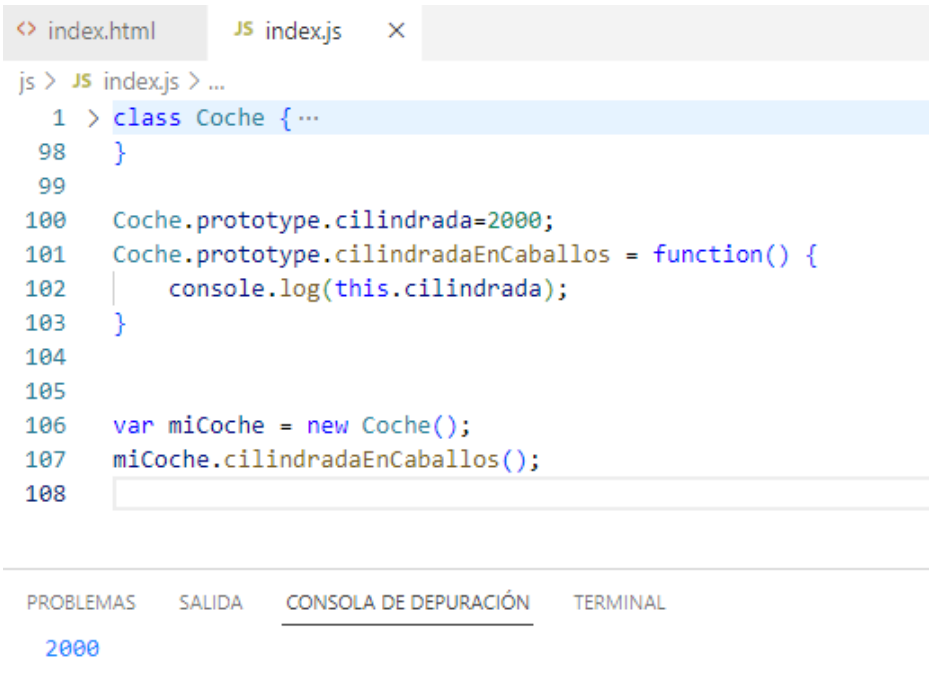
Con la clase **Coche** declarada en un ejemplo anterior

```
class Coche { ...
}
// método estático de la clase
Coche.despedir = () => {
   console.log("Adios ...");
}

// propiedad estática de la clase
Coche.GT="GT";

var miCoche = new Coche();

// propiedad y método particular de la instancia miCoche
miCoche.logo="ferrari";
miCoche.mostrar= function() {
   console.log(this.logo);
}

Coche.despedir();
miCoche.mostrar();
```

Una vez definida la clase, **no es posible el acceso a las propiedades y métodos privados** de la misma desde los métodos añadidos. Por ejemplo, en el método **mostrar** no es posible acceder a **#matrículaPrivada**.

Todas las clases de objetos tienen una propiedad llamada **prototype** que apunta a lo que serían los planos de la misma. Si añadimos una propiedad o un método a esta propiedad los estaremos haciendo a todas las instancias de la misma.

```
Coche.prototype.cilindrada=2000;
Coche.prototype.cilindradaEnCaballos = function() {
   console.log(this.cilindrada);
}

var miCoche = new Coche();
miCoche.cilindradaEnCaballos();
```



Esto se puede hacer con las clases nativas del lenguaje o del **DOM,** en general con todas las clases que ya estén definidas**.** Por ejemplo podríamos añadir a la clase **String** un método que devolviera la cadena invertida:

```
String.prototype.invert = function() {
   let aux="";
   for (let i=0; i<this.length;i++)
      aux=this[i]+aux;
   return aux;
}

console.log("Salamanca".invert());
```

5 - Blink

Vamos a desarrollar un método, que añadiremos a todos los elemento de imagen de una página, que haga efecto parpadeo. El método tendrá un argumento, será un número entero e indicará el número de milisegundos entre parpadeo. Si es **0** el parpadeo que pudiera haber se detendrá.

```
HTMLImageElement.prototype.__temporizador;
HTMLImageElement.prototype.blink = function (intervalo) {
   var intervaloBlink = 0;
   clearTimeout(this.__temporizador);
   // validación de intervalo
   if ((intervalo != undefined) && !/^(0|[1-9][0-9]{0,8})$/.test(intervalo))
      throw new RangeError("valor de intervalo incorrecto");
   intervaloBlink = parseInt(intervalo);
```

```
    let parpadea = () => {
      if (intervaloBlink == 0)
        this.style.visibility = "visible";
      else {
        this.style.visibility = (this.style.visibility == "visible") ? "hidden" : "visible";
        this.__temporizador = setTimeout(parpadea, intervaloBlink);
      }
    }
    parpadea();
}
```

Se ha añadido al **proptype** de la clase **HTMLImageElement** que es la clase que el **DOM** instancia cuando aparece un elemento **IMG** en el código html.

El método **setTimeout** del objeto **window** difiere la ejecución que se pasa como primer argumento en el número de milisegundo que se pasa como segundo argumento. Este método devuelve un manejador, **handler**, al temporizador puesto que permite cancelarlo si está en vigor haciendo uso del método **clearTimeout**. Esta ejecución diferida solo se ejecuta una vez. Si quisiéramos ejecutar de forma periódica el código de la función podríamos utilizar el método **setInterval** para activarlo y **clearInterval** para cancelarlo. Cuando no se conoce exactamente el tiempo que tarda la función en su ejecución es conveniente utilizar **setTimeout** y al final del código de la función volver a ejecutar **setTimeout** así nos aseguramos que no se solapan ejecuciones consecutivas de la función. El código propuesto ha seguido este criterio.

Aunque parezca inútil el uso de una variable para almacenar el **handler** del temporizador y la ejecución de **clearTimeout,** porque puede parecer que solo se va a volver a ejecutar el código cuando el anterior timeout ha terminado, no lo es. Piénsese que ocurriría si se ejecutasen varios **blink** de forma seguida.

Dado que no se tiene acceso a las variables privadas de las clases ya definidas y que tampoco se pueden definir variables privadas nuevas, lo que se ve en el código **__temporizador** es una propiedad pública. Se ha comenzado con **__** para indicar que aunque sea una propiedad pública no debe hacerse uso de ella en otro código. Viene a ser como una declaración para otros programadores de que se trata de una propiedad privada, aunque no lo sea. Tampoco podríamos haber llamado **#temporizador** ya que el uso de **#** se limita al ámbito de una declaración **class**.

Se debe tener especial cuidado de no incluir referencia a **this** entre los argumentos de la función de **callback** de los métodos **setTimeout** y **setInterval** ya que la resolución de **this** cuando se lance la ejecución diferida de la función no apuntará al objeto en cuestión si no que apuntará al objeto **window**. Esto se soluciona no utilizando referencias a **this** sino a funciones y propiedades privadas de la función que lanza el timeout. En el ejemplo anterior, si quisiéramos tener una propiedad pública que nos permitiera indicar el número de milisegundos, lo que haríamos sería envolver esta propiedad en una variable privada que es la que pasaríamos al temporizador:

```
HTMLImageElement.prototype.__temporizador;
HTMLImageElement.prototype.intervalo=100;
HTMLImageElement.prototype.blink = function () {
  var intervaloBlink = parseInt(this.intervalo);
  clearTimeout(this.__temporizador);
  let parpadea = () => {
    if (intervaloBlink == 0)
```

```
            this.style.visibility = "visible";
        else {
            this.style.visibility = (this.style.visibility == "visible") ? "hidden" : "visible";
            this.__temporizador = setTimeout(parpadea, intervaloBlink);
        }
    }
    parpadea();
}
```

Si añadimos el control de la propiedad quedaría:

```
HTMLImageElement.prototype.__temporizador;
HTMLImageElement.prototype.intervalo = 100;
HTMLImageElement.prototype.blink = function () {
    // validación de intervalo
    if (!/^(0|[1-9][0-9]{0,8})$/.test(this.intervalo))
        throw new RangeError("valor de intervalo incorrecto");
    var intervaloBlink = parseInt(this.intervalo);
    clearTimeout(this.__temporizador);
    let parpadea = () => {
        if (intervaloBlink == 0)
            this.style.visibility = "visible";
        else {
            this.style.visibility = (this.style.visibility == "visible") ? "hidden" : "visible";
            this.__temporizador = setTimeout(parpadea, intervaloBlink);
        }
    }
    parpadea();
}
```

Un ejemplo de uso completo de este método y propiedad podría ser:

index.html

```
<!DOCTYPE html>
<html lang="es">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Blink</title>
    <link rel="stylesheet" href="css/index.css">
</head>

<body>
    <div>
        <img src="images/1.jpg">
    </div>
    <div>
        <img src="images/2.jpg">
    </div>
    <script src="js/imageEffects.js"></script>
    <script src="js/index.js"></script>
</body>

</html>
```

index.css

```
img {
  width: 200px;
}
```

index.js

```
var divs = document.querySelectorAll("div");
for (let i of [0, 1]) {
   divs[i].parpadeando = false;
   divs[i].addEventListener("click", () => {
      divs[i].parpadeando = (!divs[i].parpadeando);
      let img = divs[i].querySelector("IMG");
      img.intervalo = (divs[i].parpadeando) ? 100 : 0;
      img.blink();
   })
};
```

imageEffects.js

```
HTMLImageElement.prototype.__temporizador;
HTMLImageElement.prototype.intervalo = 100;
HTMLImageElement.prototype.blink = function () {
   // validación de intervalo
   if (!/^(0|[1-9][0-9]{0,8})$/.test(this.intervalo))
      throw new RangeError("valor de intervalo incorrecto");
   var intervaloBlink = parseInt(this.intervalo);
   clearTimeout(this.__temporizador);
   let parpadea = () => {
      if (intervaloBlink == 0)
         this.style.visibility = "visible";
      else {
         this.style.visibility = (this.style.visibility == "visible") ? "hidden" : "visible";
         this.__temporizador = setTimeout(parpadea, intervaloBlink);
      }
   }
   parpadea();
}
```

6 - Filtrar teclas

En el siguiente código se añade un nuevo atributo a todos los **input** de texto llamado **chars** que admite con sintaxis de patrón expresión regular los caracteres que se permite introducir en el **input**.

index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Filtrar caracteres</title>
</head>
<body>
    <input type="text" chars="a-z0-9">
    <input type="text" chars="0-9BCDFGHJKLMNPQRSTVWXZ">
    <script src="js/index.js"></script>
</body>
</html>
```

index.js

```
var inputs= document.querySelectorAll("input[type=text][chars]");
for (elemento of inputs)
{
    elemento.addEventListener("keypress", e =>
    {
        var expresion = new RegExp("["+e.target.getAttribute("chars")+"]");
        if (! (expresion.test(e.key)))
            e.preventDefault();
    })
}
```

Para acceder a un atributo no estándar tenemos el método **getAttribute** de todos los nodos **DOM**. Mediante **querySelectorAll** recuperamos todos los **input** de tipo **text** que tengan puesto el atributo **chars**. Con **new Regexp** construimos una expresión regular suministrando la cadena que iría entre **/** en la expresión literal, y como solo se captura en el **keypress** un carácter, en **chars** se enumeran con sintaxis de expresión regular los caracteres permitidos, es por lo que se encierra entre corchetes **[]**.

En el ejemplo, en el primer **input,** solo se admiten letras minúsculas del alfabeto inglés y dígitos, y en el segundo dígitos y letras mayúsculas del alfabeto inglés sin las vocales.