UD01- Introducción a Javascript

UD01- Introducción a Javascript

1. Introducción
    - 1.1 Un poco de historia
    - 1.2 Soporte en los navegadores
    - 1.3 Herramientas
        - 1.3.1 La consola del navegador
        - 1.3.2 Editores
        - 1.3.3 Editores on-line
        - 1.3.4 npm
        - 1.3.5 git
    - 1.4 Incluir javascript en una página web
    - 1.5 Mostrar información
2. Sintaxis
    - 2.1 Variables
    - 2.2 Funciones
        - 2.2.1 Parámetros
        - 2.2.2 Funciones anónimas
        - 2.2.3 Arrow functions (funciones *lambda*)
    - 2.3 Estructuras y bucles
        - 2.3.1 Estructura condicional: if
        - 2.3.2 Estructura condicional: switch
        - 2.3.3 Bucle *while*
        - 2.3.4 Bucle: for
            - 2.3.4.1 Bucle: for con contador
            - 2.3.4.2 Bucle: for...in
            - 2.3.4.3 Bucle: for...of
    - 2.4 Tipos de datos básicos
        - 2.4.1 Casting de variables
        - 2.4.2 Number
        - 2.4.3 String
            - 2.4.3.1 Template literals
        - 2.4.4 Boolean
3. Manejo de errores
4. Buenas prácticas
    - 4.1 'use strict'

4.2 Variables
4.3 Errores
4.4 Otras
4.5 Clean Code
Bibliografía

---

1. Introducción

En las páginas web el elemento fundamental es el fichero HTML con la información a mostrar en el navegador. Posteriormente surgió la posibilidad de "decorar" esa información para mejorar su apariencia, lo que dio lugar al CSS. Y también se pensó en dar dinamismo a las páginas y apareció el lenguaje Javascript.

En un primer momento las 3 cosas estaban mezcladas en el fichero HTML pero eso complicaba bastante el poder leer esa página a la hora de mantenerla por lo que se pensó en separar los 3 elementos básicos:

- HTML: se encarga de estructurar la página y proporciona su información, pero es una información estática
- CSS: es lo que da forma a dicha información, permite mejorar su apariencia, permite que se adapte a distintos dispositivos, ...
- Javascript: es el que da vida a un sitio web y le permite reaccionar a las acciones del usuario

Por tanto nuestras aplicaciones tendrán estos 3 elementos y lo recomendable es que estén separados en distintos ficheros:

- El HTML lo tendremos habitualmente en un fichero index.html, normalmente en una carpeta llamada *public*
- El CSS lo tendremos en uno o más ficheros con extensión *.css* dentro de una carpeta llamada *styles*
- EL JS estará en ficheros con extensión *.js* en un directorio llamado *scripts*

Las características principales de Javascript son:

- Es un lenguaje interpretado, no compilado
- Se ejecuta en el lado cliente (en un navegador web), aunque hay implementaciones como NodeJS para el lado servidor
- Es un lenguaje orientado a objetos (podemos crear e instanciar objetos y usar objetos predefinidos del lenguaje) pero basado en prototipos (por debajo un objeto es un prototipo y nosotros podemos crear objetos sin instanciarlos, haciendo copias del prototipo)
- Se trata de un lenguaje débilmente tipado, con tipificación dinámica (no se indica el tipo de datos de una variable al declararla e incluso puede cambiarse)

Lo usaremos para:

- Cambiar el contenido de la página
- Cambiar los atributos de un elemento
- Cambiar la apariencia de algo
- Validar datos de formularios
- ...

Sin embargo, por razones de seguridad, Javascript no nos permite hacer cosas como:

- Acceder al sistema de ficheros del cliente
- Capturar datos de un servidor (puede pedirlo y el servidor se los servirá, o no)
- Modificar las preferencias del navegador
- Enviar e-mails de forma invisible o crear ventanas sin que el usuario lo vea
- ...

1.1 Un poco de historia

Javascript es una implementación del lenguaje **ECMAScript** (el estándar que define sus características). El lenguaje surgió en 1997 y todos los navegadores a partir de 2012 soportan al menos la versión **ES5.1** completamente. En 2015 se lanzó la 6ª versión, inicialmente llamada **ES6** y posteriormente renombrada como **ES2015**, que introdujo importantes mejoras en el lenguaje y que es la versión que usaremos nosotros. Desde entonces van saliendo nuevas versiones cada año que introducen cambios pequeños. La última es la **ES2022** aprobada en Junio de 2022.

Las principales mejoras que introdujo ES2015 son: clases de objetos, let, for..of, Map, Set, Arrow functions, Promesas, spread, destructuring, ...

1.2 Soporte en los navegadores

Los navegadores no se adaptan inmediatamente a las nuevas versiones de Javascript por lo que puede ser un problema usar una versión muy moderna ya que puede haber partes de los programas que no funcionen en los navegadores de muchos usuarios. En la página de *Kangax* podemos ver la compatibilidad de los diferentes navegadores con las distintas versiones de Javascript. También podemos usar *CanIUse* para buscar la compatibilidad de un elemento concreto de Javascript así como de HTML5 o CSS3.

Si queremos asegurar la máxima compatibilidad debemos usar la versión ES5 (pero nos perdemos muchas mejoras del lenguaje) o mejor, usar la ES6 (o posterior) y después *transpilar* nuestro código a la version ES5. De esto se ocupan los *transpiladores* (**Babel** es el más conocido) por lo que no suponen un esfuerzo extra para el programador.

1.3 Herramientas

1.3.1 La consola del navegador

Es la herramienta que más nos va a ayudar a la hora de depurar nuestro código. Abrimos las herramientas para el desarrollador (en Chrome y Firefox pulsando la tecla *F12*) y vamos a la pestaña *Consola*:

Allí vemos mensajes del navegador como errores y advertencias que genera el código y todos los mensajes que pongamos en el código para ayudarnos a depurarlo (cusando los comandos **console.log** y **console.error**).

Además en ella podemos escribir instrucciones Javascript que se ejecutarán mostrando su resultado. También la usaremos para mostrar el valor de nuestras variables y para probar código que, una vez que funcione correctamente, lo copiaremos a nuestro programa.

> EJERCICIO: abre la consola y prueba las funciones *alert*, *confirm* y *prompt*.

Siempre depuraremos los programas desde aquí (ponemos puntos de interrupción, vemos el valor de las variables, ...).

1.3.2 Editores

Podemos usar el que más nos guste, desde editores tan simples como NotePad++ hasta complejos IDEs. La mayoría soportan las últimas versiones de la sintaxis de Javascript (Netbeans, Eclipse, Visual Studio, Sublime, Atom, Kate, Notepad++, ...). Vamos a utilizar **Visual Studio Code** por su sencillez y por los plugins que incorpora para hacer más cómodo el trabajo. En *Visual Studio Code* instalaremos algún *plugin* como:

- SonarLint: es más que un *linter* e informa de todo tipo de errores pero también del código que no cumple las recomendaciones (incluye gran número de reglas). Marca el código mientras lo escribimos y además podemos ver todas las advertencias en el panel de Problemas.
- Vetur: lo instalaremos en el segundo bloque. Necesario para trabajar con los ficheros de *Vue*

1.3.3 Editores on-line

Son muy útiles porque permiten ver el código y el resultado a la vez. Normalmente tienen varias pestañas o secciones de la página donde poner el código HTML, CSS yJavascript y ver su resultado.

Algunos de los más conocidos son Codesandbox, Fiddle, Plunker, CodePen, ...aunque hay muchos más.

Ejemplo de 'Hello World' en CodePen:

1.3.4 npm

npm es el gestor de paquetes del framework Javascript **Node.js** y suele utilizarse en programación *frontend* como gestor de dependencias de la aplicación. Esto significa que será la herramienta que se encargará de descargar y poner a disposición de nuestra aplicación todas las librerías Javascript que vayamos a utilizar, por ejemplo para hacer los tests de nuestros programas (en algunas prácticas haremos o pasaremos tests unitarios ya hechos para comprobar que nuestros programas funcionan correctamente y, sobre todo, que continúan haciéndolo tras realizar alguna modificación en ellos).

Para instalar *npm* tenemos que instalar *NodeJS*. Podemos instalarlo desde los repositorios como cualquier otro programa (`apt install nodejs`), pero posiblemente nos instalará una versión poco actualizada por lo que es mejor instalarlo desde NodeSource siguiendo las instrucciones que allí se indican, que básicamente son ejecutar:

```
curl -sL https://deb.nodesource.com/setup_X.y | sudo -E bash -
sudo apt-get install -y nodejs
```

(cambiaremos X.y por la versión que queramos, normalmente la últma versión estable).

También podemos descargarlo directamente desde NodeJS.org, descomprimir el paquete e instalarlo (`dpkg -i _nombre_del_paquete_`).

Con eso ya tendremos *npm* en nuestro equipo. Podemos comprovar la versión que tenemos con:

```
npm -v
```

1.3.5 git

Usaremos repositorios para poder realizar el control de versiones. Para instalarlo simplemente habrá que instalar el paquete *git* (`apt-get install git`).

Podemos gestionar git desde la consola o se puede instal·ar una extensión per a utilizarlo desde el entorno de programación. En el caso de Visual Studio lo encontramos en https://code.visualstudio.com/docs/editor/versioncontrol.

1.4 Incluir javascript en una página web

El código Javascript va entre etiquetas <*script*>. Puede ponerse en el <*head*> o en el <*body*>. Funciona como cualquier otra etiqueta y el navegador la interpreta cuando llega a ella (va leyendo y ejecutando el fichero línea a línea). Podéis ver en este vídeo un ejemplo muy simple de cómo se ejecuta el código en el HEAD y en el BODY.

Lo mejor en cuanto a rendimiento es ponerla al final del <*body*> para que no se detenga el renderizado de la página mientras se descarga y se ejecuta el código. También podemos ponerlo en el <head> pero usando los atributos **async** y/o **defer** (en Internet encontraréis mucha información sobre esta cuestión, por ejemplo aquí).

Como se ve en el primer vídeo, es posible poner el código directamente entre la etiqueta <*script*> y su etiqueta de finalización pero lo correcto es que esté en un fichero externo (con extensión **.js**) que cargamos mediante el atributo *src* de la etiqueta. Así conseguimos que la página HTML cargue más rápido (si lo ponemos al final del BODY o usamos *async*) y además no mezclar HTML y JS en el mismo fichero, lo mejora la legibilidad del código y facilita su mantenimiento:

```
<script src="./scripts/main.js"></script>
```

1.5 Mostrar información

Javascript permite mostrar al usuario ventanas modales para pedirle o mostrarle información. Las funciones que lo hacen son:

- `window.alert(mensaje)`: Muesta en una ventana modal *mensaje* con un botón de *Aceptar* para cerra la ventana.
- `window.confirm(mensaje)`: Muesta en una ventana modal *mensaje* con botones de *Aceptar* y *Cancelar*. La función devuelve **true** o **false** en función del botón pulsado por el usuario.
- `window.prompt(mensaje [, valor predeterminado])`: Muesta en una ventana modal *mensaje* y debajo tiene un campo donde el usuario puede escribir, junto con botones de *Aceptar* y *Cancelar*. La función devuelve el valor introducido por el usuario como texto (es decir que si introduce 54 lo que se obtiene es "54") o **false** si el usuario pulsa *Cancelar*.

También se pueden escribir las funciones sin *window.* (es decir `alert('Hola')` en vez de `window.alert('Hola')`) ya que en Javascript todos los métodos y propiedades de los que no se indica de qué objeto son se ejecutan en el objeto *window*.

Si queremos mostrar una información para depurar nuestro código no utilizaremos *alert(mensaje)* sino `console.log(mensaje)` o `console.error(mensaje)`. Estas funciones muestran la información pero en la consola del navegador. La diferencia es que *console.error* la muestra como si fuera un error de Javascript.

2. Sintaxis

2.1 Variables

Javascript es un lenguaje débilmente tipado. Esto significa que no se indica de qué tipo es una variable al declararla e incluso puede cambiar su tipo a lo largo de la ejecución del programa. Ejemplo:

```
let miVariable;          // declaro miVariable y como no se asigno un valor valdrá undefined
miVariable='Hola';       // ahora su valor es 'Hola', por tanto contiene una cadena de texto
miVariable=34;           // pero ahora contiene un número
miVariable=[3, 45, 2];   // y ahora un array
miVariable=undefined;    // para volver a valer el valor especial undefined
```

> EJERCICIO: Ejecuta en la consola del navegador las instrucciones anteriores y comprueba el valor de miVariable tras cada instrucción (para ver el valor de una variable simplemente ponemos en la consola su nombre: `miVariable`

Ni siquiera estamos obligados a declarar una variable antes de usarla, aunque es recomendable para evitar errores que nos costará depurar. Podemos hacer que se produzca un error si no declaramos una variable incluyendo al principio de nuestro código la instrucción

```
'use strict'
```

Las variables de declaran con **let** (lo recomendado desde ES2015), aunque también pueden declararse con **var**. La diferencia es que con *let* la variable sólo existe en el bloque en que se declara mientras que con *var* la variable existe en toda la función en que se declara:

```
if (edad < 18) {
  let textoLet = 'Eres mayor de edad';
  var textoVar = 'Eres mayor de edad';
} else {
  let textoLet = 'Eres menor de edad';
  var textoVar = 'Eres menor de edad';
}
console.log(textoLet);  // mostrará undefined porque fuera del if no existe la variable
console.log(textoVar);  // mostrará la cadena
```

Cualquier variable que no se declara dentro de una función (o si se usa sin declarar) es *global*. Debemos siempre intentar NO usar variables globales.

Se recomienda que Los nombres de las variables sigan la sintaxis *camelCase* (ej.: *miPrimeraVariable*).

Desde ES2015 también podemos declarar constantes con **const**. Se les debe dar un valor al declararlas y si intentamos modificarlo posteriorment se produce un error.

NOTA: en la página de Babel podemos teclear código en ES2015 y ver cómo quedaría una vez transpilado a ES5.

2.2 Funciones

Se declaran con **function** y se les pasan los parámetros entre paréntesis. La función puede devolver un valor usando **return** (si no tiene *return* es como si devolviera *undefined*).

Puede usarse una función antes de haberla declarado por el comportamiento de Javascript llamado *hoisting*: el navegador primero carga todas las funciones y mueve las declaraciones de las variables al principio y luego ejecuta el código.

> EJERCICIO: Haz una función que te pida que escribas algo y muestre un alert diciendo 'Has escrito...' y el valor introducido. Pruébala en la consola (pegas allí la función y luego la llamas desde la consola)

2.2.1 Parámetros

Si se llama una función con menos parámetros de los declarados el valor de los parámetros no pasados será *undefined*:

```
function potencia(base, exponente) {
    console.log(base);            // muestra 4
    console.log(exponente);       // muestra undefined
    let valor=1;
    for (let i=1; i<=exponente; i++) {
      valor=valor*base;
    }
    return valor;
}

potencia(4);    // devolverá 1 ya que no se ejecuta el for
```

Podemos dar un **valor por defecto** a los parámetros por si no los pasan asignándoles el valor al definirlos:

```
function potencia(base, exponente=2) {
    console.log(base);            // muestra 4
    console.log(exponente);       // muestra 2 la primera vez y 5 la segunda
    let valor=1;
    for (let i=1; i<=exponente; i++) {
      valor=valor*base;
    }
    return valor;
}

console.log(potencia(4));         // mostrará 16 (4^2)
console.log(potencia(4,5));       // mostrará 1024 (4^5)
```

NOTA: En ES5 para dar un valor por defecto a una variable se hacía

```
function potencia(base, exponente) {
exponente = exponente || 2;       // si exponente vale undefined se la asigna el valor 2
 ...
```

También es posible acceder a los parámetros desde el array **arguments[]** si no sabemos cuántos recibiremos:

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

console.log(suma(4, 2));                    // mostrará 6
console.log(suma(4, 2, 5, 3, 2, 1, 3));     // mostrará 20
```

En Javascript las funciones son un tipo de datos más por lo que podemos hacer cosas como pasarlas por argumento o asignarlas a una variable:

```
const cuadrado = function(value) {
  return value * value
}
function aplica_fn(dato, funcion_a_aplicar) {
    return funcion_a_aplicar(dato);
}

aplica_fn(3, cuadrado);     // devolverá 9 (3^2)
```

A este tipo de funciones se llama *funciones de primera clase* y son típicas de lenguajes funcionales.

2.2.2 Funciones anónimas

Como acabamos de ver podemos definir una función sin darle un nombre. Dicha función puede asignarse a una variable, autoejecutarse o asignasrse a un manejador de eventos. Ejemplo:

```
let holaMundo = function() {
    alert('Hola mundo!');
}

holaMundo();        // se ejecuta la función
```

Como vemos asignamos una función a una variable de forma que podamos "ejecutar" dicha variable.

2.2.3 Arrow functions (funciones *lambda*)

ES2015 permite declarar una función anónima de forma más corta. Ejemplo sin *arrow function*:

```
let potencia = function(base, exponente) {
    let valor=1;
    for (let i=1; i<=exponente; i++) {
      valor=valor*base;
    }
    return valor;
}
```

Al escribirla con la sintaxis de una *arrow function* lo que hacemos es:

- Eliminamos la palabra *function*
- Si sólo tiene 1 parámetro podemos eliminar los paréntesis de los parámetros
- Ponemos el símbolo =>
- Si la función sólo tiene 1 línea podemos eliminamr las { } y la palabra *return*

El ejemplo con *arrow function*:

```
let potencia = (base, exponente) => {
    let valor=1;
    for (let i=1; i<=exponente; i++) {
      valor=valor*base;
    }
    return valor;
}
```

Otro ejemplo, sin *arrow function*:

```
let cuadrado = function(base) {
    return base * base;
}
```

conn *arrow function*:

```
let cuadrado = base => base * base;
```

EJERCICIO: Haz una *arrow function* que devuelva el cubo del número pasado como parámetro y pruébala desde la consola. Escríbela primero en la forma habitual y luego la "traduces" a *arrow function*.

2.3 Estructuras y bucles

2.3.1 Estructura condicional: if

El **if** es como en la mayoría de lenguajes. Puede tener asociado un **else** y pueden anidarse varios con **else if**.

```
if (condicion) {
    ...
} else if (condicion2) {
    ...
} else if (condicion3) {
    ...
} else {
    ...
}
```

Ejemplo:

```
if (edad < 18) {
    console.log('Es menor de edad');
} else if (edad > 65) {
    console.log('Está jubilado');
} else {
    console.log('Edad correcta');
}
```

Se puede usar el operador **? :** que es como un *if* que devuelve un valor:

```
let esMayorDeEdad = (edad>18)?true:false;
```

2.3.2 Estructura condicional: switch

El **switch** también es como en la mayoría de lenguajes. Hay que poner *break* al final de cada bloque para que no continúe evaluando:

```
switch(color) {
    case 'blanco':
    case 'amarillo':    // Ambos colores entran aquí
        colorFondo='azul';
        break;
    case 'azul':
        colorFondo='amarillo';
        break;
    default:            // Para cualquier otro valor
        colorFondo='negro';
}
```

Javascript permite que el *switch* en vez de evaluar valores pueda evaluar expresiones. En este caso se pone como condición *true*:

```
switch(true) {
    case age < 18:
        console.log('Eres muy joven para entrar');
        break;
    case age < 65:
        console.log('Puedes entrar');
        break;
    default:
        console.log('Eres muy mayor para entrar');
}
```

2.3.3 Bucle *while*

Podemos usar el bucle *while...do*

```
while (condicion) {
    // sentencias
}
```

que se ejecutará 0 o más veces. Ejemplo:

```
let nota=prompt('Introduce una nota (o cancela para finalizar)');
while (nota) {
    console.log('La nota introducida es: '+nota);
    nota=prompt('Introduce una nota (o cancela para finalizar)');
}
```

O el bucle *do...while*:

```
do {
    // sentencias
} while (condicion)
```

que al menos se ejecutará 1 vez. Ejemplo:

```
let nota;
do {
    nota=prompt('Introduce una nota (o cancela para finalizar)');
    console.log('La nota introducida es: '+nota);
} while (nota)
```

EJERCICIO: Haz un programa para que el usuario juegue a adivinar un número. Obtén un número al azar (busca por internet cómo se hace o simplemente guarda el número que quieras en una variable) y ve pidiendo al usuario que introduzca un número. Si es el que busca le dices que lo ha encontrado y si no le mostrarás si el número que busca el mayor o menor que el introducido. El juego acaba cuando el usuario encuentra el número o cuando pulsa en 'Cancelar' (en ese caso le mostraremos un mensaje de que ha cancelado el juego).

2.3.4 Bucle: for

Tenemos muchos *for* que podemos usar.

2.3.4.1 Bucle: for con contador

Creamos una variable contador que controla las veces que se ejecuta el for:

```
let datos=[5, 23, 12, 85]
let sumaDatos=0;

for (let i=0; i<datos.length; i++) {
    sumaDatos += datos[i];
}
// El valor de sumaDatos será 125
```

EJERCICIO: El factorial de un número entero n es una operación matemática que consiste en multiplicar ese número por todos los enteros menores que él: **n x (n-1) x (n-2) x ... x 1**. Así, el factorial de 5 (se escribe 5!) vale **5! = 5 x 4 x 3 x 2 x 1 = 120**. Haz un script que calcule el factorial de un número entero.

2.3.4.2 Bucle: for...in

El bucle se ejecuta una vez para cada elemento del array (o propiedad del objeto) y se crea una variable contador que toma como valores la posición del elemento en el array:

```
let datos=[5, 23, 12, 85]
let sumaDatos=0;

for (let indice in datos) {
    sumaDatos += datos[indice];     // los valores que toma indice son 0, 1, 2, 3
}
// El valor de sumaDatos será 125
```

También sirve para recorrer las propiedades de un objeto:

```
let profe={
    nom:'Juan',
    ape1='Pla',
    ape2='Pla'
}
let nombre='';

for (var campo in profe) {
   nombre += profe.campo + ' '; // o profe[campo];
}
// El valor de nombre será 'Juan Pla Pla '
```

2.3.4.3 Bucle: for...of

Es similar al *for...in* pero la variable contador en vez de tomar como valor cada índice toma cada elemento. Es nuevo en ES2015:

```
let datos = [5, 23, 12, 85]
let sumaDatos = 0;

for (let valor of datos) {
    sumaDatos += valor;       // los valores que toma valor son 5, 23, 12, 85
}
// El valor de sumaDatos será 125
```

También sirve para recorrer los caracteres de una cadena de texto:

```
let cadena = 'Hola';

for (let letra of cadena) {
    console.log(letra);     // los valores de letra son 'H', 'o', 'l', 'a'
}
```

EJERCICIO: Haz 3 funciones a las que se le pasa como parámetro un array de notas y devuelve la nota media. Cada una usará un for de una de las 3 formas vistas. Pruébalas en la consola

2.4 Tipos de datos básicos

Para saber de qué tipo es el valor de una variable tenemos el operador **typeof**. Ej.:

- `typeof 3` devuelve *number*
- `typeof 'Hola'` devuelve *string*

En Javascript hay 2 valores especiales:

- **undefined**: es lo que vale una variable a la que no se ha asignado ningún valor
- **null**: es un tipo de valor especial que podemos asignar a una variable. Es como un objeto vacío (`typeof null` devuelve *object*)

También hay otros valores especiales relacionados con operaciones con números:

- **NaN** (*Not a Number*): indica que el resultado de la operación no puede ser convertido a un número (ej. `'Hola'*2`, aunque `'2'*2` daría 4 ya que se convierte la cadena '2' al número 2)
- **Infinity** y **-Infinity**: indica que el resultado es demasiado grande o demasiado pequeño (ej. `1/0` o `-1/0`)

2.4.1 Casting de variables

Como hemos dicho las variables pueden contener cualquier tipo de valor y, en las operaciones, Javascript realiza **automáticamente** las conversiones necesarias para, si es posible, realizar la operación. Por ejemplo:

- `'4' / 2` devuelve 2 (convierte '4' en 4 y realiza la operación)
- `'23' - null` devuelve 0 (hace 23 - 0)
- `'23' - undefined` devuelve *NaN* (no puede convertir undefined a nada así que no puede hacer la operación)
- `'23' * true` devuelve 23 (23 * 1)
- `'23' * 'Hello'` devuelve *NaN* (no puede convertir 'Hello')

- `23 + 'Hello'` devuelve '23Hello' (+ es el operador de concatenación así que convierte 23 a '23' y los concatena)
- `23 + '23'` devuelve 2323 (OJO, convierte 23 a '23', no al revés)

Además comentar que en Javascript todo son ojetos por lo que todo tiene métodos y propiedades. Veamos brevemente los tipos de datos básicos.

> EJERCICIO: Prueba en la consola las operaciones anteriores y alguna más con la que tengas dudas de qué devolverá

2.4.2 Number

Sólo hay 1 tipo de números, no existen enteros y decimales. El tipo de dato para cualquier número es **number**. El carácter para la coma decimal es el **.** (como en inglés, así que 23,12 debemos escribirlo como 23.12).

Tenemos los operadores aritméticos **+**, **-**, *****, **/** y **%** y los unarios **++** y **--** y existen los valores especiales **Infinity** y **-Infinity** (`23 / 0` no produce un error sino que devuelve *Infinity*).

Podemos usar los operadores artméticos junto al operador de asignación **=** (+=, -=, *=, /= y %=).

Algunos métodos útiles de los números son:

- **.toFixed(num)**: redondea el número a los decimales indicados. Ej. `23.2376.toFixed(2)` devuelve 23.24
- **.toLocaleString()**: devuelve el número convertido al formato local. Ej. `23.76.toLocaleString()` devuelve '23,76' (convierte el punto decimal en coma)

Podemos forzar la conversión a número con la función **Number(valor)**. Ejemplo `Number('23.12')`devuelve 23.12

Otras funciones útiles son:

- **isNaN(valor)**: nos dice si el valor pasado es un número (false) o no (true)
- **isFinite(valor)**: devuelve *true* si el valor es finito (no es *Infinity* ni *-Infinity*).
- **parseInt(valor)**: convierte el valor pasado a un número entero. Siempre que compience por un número la conversión se podrá hacer. Ej.:

```
parseInt(3.65)       // Devuelve 3
parseInt('3.65')     // Devuelve 3
parseInt('3 manzanas')    // Devuelve 3, Number devolvería NaN
```

- **parseFloat(valor)**: como la anterior pero conserva los decimales

OJO: al sumar floats podemos tener problemas:

```
console.log(0.1 + 0.2)     // imprime 0.30000000000000004
```

Para evitarlo redondead los resultados (o `(0.1*10 + 0.2*10) / 10`).

> EJERCICIO: Modifica la funciónque quieras de calcular la nota media para que devuelva la media con 1 decimal

> EJERCICIO: Modifica la función que devuelve el cubo de un número para que compruebe si el parámetro pasado es un número entero. Si no es un entero o no es un número mostrará un alert indicando cuál es el problema yndevolverá false.

2.4.3 String

Las cadenas de texto van entre comillas simples o dobles, es indiferente. Podemos escapar un caràcter con \ (ej. `'Hola \'Mundo\''` devuelve *Hola 'Mundo'*).

Para forzar la conversión a cadena se usa la función **String(valor)** (ej. `String(23)` devuelve '23')

El operador de concatenación de cadenas es **+**. Ojo porque si pedimos un dato con *prompt* siempre devuelve una cadena así que si le pedimos la edad al usuario (por ejemplo 20) y se sumamos 10 tendremos 2010 ('20'+10).

Algunos métodos y propiedades de las cadenas son:

- **.length**: devuelve la longitud de una cadena. Ej.: `'Hola mundo'.length` devuelve 10
- **.charAt(posición)**: `'Hola mundo'.charAt(0)` devuelve 'H'
- **.indexOf(carácter)**: `'Hola mundo'.indexOf('o')` devuelve 1. Si no se encuentra devuelve -1
- **.lastIndexOf(carácter)**: `'Hola mundo'.lastIndexOf('o')` devuelve 9
- **.substring(desde, hasta)**: `'Hola mundo'.substring(2,4)` devuelve 'la'
- **.substr(desde, num caracteres)**: `'Hola mundo'.substr(2,4)` devuelve 'la m'
- **.replace(busco, reemplaza)**: `'Hola mundo'.replace('Hola', 'Adiós')` devuelve 'Adiós mundo'
- **.toLocaleLowerCase()**: `'Hola mundo'.toLocaleLowerCase()` devuelve 'hola mundo'
- **.toLocaleUpperCase()**: `'Hola mundo'.toLocaleUpperCase()` devuelve 'HOLA MUNDO'
- **.localeCompare(cadena)**: devuelve -1 si la cadena a que se aplica el método es anterior alfabéticamente a 'cadena', 1 si es posterior y 0 si ambas son iguales. Tiene en cuenta caracteres locales como acentos ñ, ç, etc
- **.trim(cadena)**: `'   Hola mundo   '.trim()` devuelve 'Hola mundo'
- **.startsWith(cadena)**: `'Hola mundo'.startsWith('Hol')` devuelve *true*

- **.endsWith(cadena)**: `'Hola mundo'.endsWith('Hol')` devuelve *false*
- **.includes(cadena)**: `'Hola mundo'.includes('mun')` devuelve *true*
- **.repeat(veces)**: `'Hola mundo'.repeat(3)` devuelve 'Hola mundoHola mundoHola mundo'
- **.split(sepadaror)**: `'Hola mundo'.split(' ')` devuelve el array ['Hola', 'mundo']. `'Hola mundo'.split('')` devuelve el array ['H', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o']

Podemos probar los diferentes métodos en la página de w3schools.

> EJERCICIO: Haz una función a la que se le pasa un DNI (ej. 12345678w o 87654321T) y devolverá si es correcto o no. La letra que debe corresponder a un DNI correcto se obtiene dividiendo la parte numérica entre 23 y cogiendo de la cadena 'TRWAGMYFPDXBNJZSQVHLCKE' la letra correspondiente al resto de la divisón. Por ejemplo, si el resto es 0 la letra será la T y si es 4 será la G. Prueba la función en la consola con tu DNI

2.4.3.1 Template literals

Desde ES2015 también podemos poner una cadena entre ` (acento grave) y en ese caso podemos poner dentro variables y expresiones que serán evaluadas al ponerlas dentro de **${}**. También se respetan los saltos de línea, tabuladores, etc que haya dentro. Ejemplo:

```
let edad=25;

console.log(\`El usuario tiene:
${edad} años\`)
```

Mostrará en la consola:

> El usuario tiene:

> 25 años

2.4.4 Boolean

Los valores booleanos son **true** y **false**. Para convertir algo a booleano se usar **Boolean(valor)** aunque también puede hacerse con la doble negación (**!!**). Cualquier valor se evaluará a *true* excepto 0, NaN, null, undefined o una cadena vacía ('') que se evaluarán a *false*.

Los operadores lógicos son ! (negación), && (and), || (or).

Para comparar valores tenemos **==** y **===**. La triple igualdad devuelve *true* si son igual valor y del mismo tipo. Como Javascript hace conversiones de tipos automáticas conviene usar la **===** para evitar cosas como:

- `'3' == 3` true
- `3 == 3.0` true
- `0 == false` true
- `'' == false` true
- `' ' == false` true
- `[] == false` true
- `null == false` false
- `undefined == false` false
- `undefined == null` true

También tenemos 2 operadores de *diferente*: **!=** y **!==** que se comportan como hemos dicho antes.

Los operadores relacionales son >, >=, <, <=. Cuando se compara un número y una cadena ésta se convierte a número y no al revés (`23 > '5'` devuelve *true*, aunque `'23' > '5'` devuelve *false*)

3. Manejo de errores

Si sucede un error en nuestro código el programa dejará de ejecutarse por lo que el usuario tendrá la sensación de que no hace nada (el error sólo se muestra en la consola y el usuario no suele abrirla nunca). Para evitarlo debemos intentar capturar los posibles errores de nuestro código antes de que se produzcan.

En javascript (como en muchos otros lenguajes) el manejo de errores se realiza con sentencias

```
try {
    ...
}
catch(error) {
    ...
}
```

Dentro del bloque *try* ponemos el código que queremos proteger y cualquier error producido en él será pasado al bloque *catch* donde es tratado. Opcionalmente podemos tener al final un bloque *finally* que se ejecuta tanto si se produce un error como si no. El parámetro que recibe *catch* es un objeto con las propiedades *name*, que indica el tipo de error (*SyntaxError*, *RangeError*, ... o el genérico *Error*), y *message*, que indica el texto del error producido.

En ocasiones podemos querer que nuestro código genere un error. Esto evita que tengamos que comprobar si el valor devuelto por una función es el adecuado o es un código de error. Por ejemplo tenemos una función para retirar dinero de una cuenta que recibe el saldo de la misma y la cantdad de dinero a retirar y devuelve el nuevo saldo, pero si no hay suficiente saldo no debería restar nada sino mostrar un mensaje al usuario. Sin gestión de errores haríamos:

```
function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    return false
  }
  return saldo - cantidad
}

// Y donde se llama a la función_
...
resultado = retirar(saldo, importe)
if (resultado === false
  alert('Saldo insuficiente')
} else {
  saldo = resultado
...
```

Se trata de un código poco claro que podemos mejorar lanzando un error en la función. Para ello se utiliza la instrucción `throw`:

```
 if (saldo < cantidad) {
   throw 'Saldo insuficiente'
 }
```

Por defecto al lanzar un error este será de clase *Error* pero (el código anterior es equivalente a `throw new Error('Valor no válido')`) aunque podemos lanzarlo de cualquier otra clase (`throw new RangeError('Saldo insuficiente')`) o personalizarlo.

Siempre que vayamos a ejecutar código que pueda generar un error debemos ponerlo dentro de un bloque *try* por lo que la llamada a la función que contiene el código anterior debería estar dentro de un *try*. El código del ejemplo anterior quedaría:

```
function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    throw "Saldo insuficiente"
  }
  return saldo - cantidad
}

```

```
// Siempre debemos llamar a esa función desde un bloque _try_
...
try {
  saldo = retirar(saldo, importe)
} catch(err) {
  alert(err)
}
...
```

Podemos ver en detalle cómo funcionan en la página de MDN web docs de Mozilla.

4. Buenas prácticas

Javascript nos permite hacer muchas cosas que otros lenguajes no nos dejan por lo que debemos ser cuidadosos para no cometer errores de los que no se nos va a avisar.

4.1 'use strict'

Si ponemos siempre esta sentencia al principio de nuestro código el intérprete nos avisará si usamos una variale sin declarar (muchas vecees por equivocarnos al escrbir su nombre). En concreto fuerza al navegador a no permitir:

- Usar una variable sin declarar
- Definir más de 1 vez una propiedad de un objeto
- Duplicar un parámetro en una función
- Usar números en octal
- Modificar una propiedad de sólo lectura

4.2 Variables

Algunas de las prácticas que deberíamos seguir respecto a las variables son:

- Elegir un buen nombre es fundamental. Evitar abreviaturas o nombres sin significado (a, b, c, ...)
- Evitar en lo posible variables globales
- Usar *let* para declararlas
- Usar *const* siempre que una variable no deba cambiar su valor
- Declarar todas las variables al principio
- Inicializar las variables al declararlas
- Evitar conversiones de tipo automáticas
- Usar para nombrarlas la notación *camelCase*

También es conveniente, por motivos de eficiencia no usar objetos Number, String o Boolean sino los tipos primitivos (no usar `let numero = new Number(5)` sino `let numero = 5`) y lo mismo al crear arrays, objetos o expresiones regulares (no usar `let miArray = new Array()` sino `let miArray = []`).

4.3 Errores

No se debería estar comprobando lo devuelto por una función para ver si se ha ejecutado correctamente o ha devuelto algún código de error. Si algo es erróneo no debemos devolver un código sino lanzar un error y quien ha llamado a esa función debería hacerlo dentro de un `try...catch` para capturar dicho error. Por ejemplo:

- Mal

```
function muestraDatos(datos) {
   ...
   const nombreMes = getNombreMes(datos.mes);
   if (nombreMes === null) {
       alert('El mes ' + datos.mes + ' es erróneo');
   }
   ...
}

function getNombreMes(mes) {
   mes = mes - 1; // Ajustar el número de mes al índice del array (1 = Ene, 12 = Dic)
   var meses = new Array("Ene", "Feb", "Mar", "Abr", "May", "Jun", "Jul", "Ago", "Sep", "Oct", "Nov", "Dic");
   return meses[mes];
}
```

- Bien

```
function muestraDatos(datos) {
   ...
   try {
      const nombreMes = getNombreMes(datos.mes);
   } catch(err) {
      alert(err)
   }
   ...
}

```

```
11 function getNombreMes(mes) {
12    mes = mes - 1; // Ajustar el número de mes al índice del array (1 = Ene, 12 = Dic)
13    var meses = new Array("Ene", "Feb", "Mar", "Abr", "May", "Jun", "Jul", "Ago", "Sep", "Oct", "Nov", "Dic");
14    if (meses[mes] === null) {
15        throw 'El mes ' + mes + ' es erróneo';
16    }
17    return meses[mes];
18 }
```

4.4 Otras

Algunas reglas más que deberíamos seguir son:

- Debemos ser coherentes a la hora de escribir código: por ejemplo podemos poner (recomendado) o no espacios antes y después del `=` en una asignación pero debemos hacerlo siempre igual. Existen muchas guías de estilo y muy buenas: Airbnb, Google, Idiomatic, etc. Para obligarnos a seguir las reglas podemos usar alguna herramienta *linter*.
- También es conveniente para mejorar la legibilidad de nuestro código separar las líneas de más de 80 caracteres.
- Usar `===` en las comparaciones
- Si un parámetro puede faltar al llamar a una función darle un valor por defecto
- Y para acabar **comentar el código** cuando sea necesario, pero mejor que sea lo suficientemente claro como para no necesitar comentarios

4.5 Clean Code

Estas y otras muchas recomendaciones se recogen el el libro Clean Code de *Robert C. Martin* y en muchos otros libros y articulos. Aquí tenéis un pequeño resumen traducido al castellano:

- [https://github.com/devictoribero/clean-code-javascript](

Bibliografía

- Curso 'Programación con JavaScript'. CEFIRE Xest. Arturo Bernal Mayordomo
- Curso de JavaScript y TypeScript de Arturo Bernal en Youtube
- MDN Web Docs. Moz://a. https://developer.mozilla.org/es/docs/Web/JavaScript
- Introducción a JavaScript. Librosweb. http://librosweb.es/libro/javascript/

- Curso de Javascript (Desarrollo web en entorno cliente). Ada Lovecode - Didacticode (90 vídeos)
- Apuntes Desarrollo Web en Entorno Cliente (DWEC). García Barea, Sergi
- Apuntes Desarrollo Web en Entorno Cliente (DWEC). Segura Vasco, Juan. CIPFP Batoi.