Índice

1 Introducción a Javascript

Javascript es un lenguaje del lado del cliente, por lo tanto se ejecuta en el navegador cuando cargamos una página web. Actualmente Javascript se ha expandido abarcando muchos más espacios y aplicaciones como NodeJS, nos permiten ejecutarlo en el lado del servidor.
Para poder iniciar el desarrollo con Javascript necesitamos instalar NodeJS, un editor de texto (como pueden ser el bloc de notas, Visual Studio Code, Kate,...) y mucha paciencia.

2 Instalando NodeJS

Para instalarlo se irá a la página:
https://nodejs.org/
y se descargará el instalador correspondiente.

Para hacer nuestras pruebas se escribirá nuestro código usando nuestro editor de texto favorito y se guardará con la extensión ".js". Por ejemplo:
hola_mundo.js

```
console.log("Hola mundo");
```

Para ejecutarlo se escribirá en la consola:

```
node hola_mundo.js
```

Nos aparecerá en consola:

```
$ node hola_mundo.js
Hola mundo
```

3 Cometarios

A veces nos interesará introducir comentarios en nuestro código. Estos comentarios son ignorados por el navegador, pero nos ayudarán mucho a entender el código o cómo se realiza una determinada tarea. Hay dos formas de introducir los comentarios:

La primera forma es usando //. Los comentarios con // comienzan donde se introduce // y acaban en el final de la línea.

```
// Este es un comentario
console.log('Hola mundo');  // Introduce 'Hola mundo' en la página
// La siguiente línea no se ejecutará pues está comentada:
// console.log('Adios mundo');
```

Otra forma de introducir los comentarios es usando /* y */. Los comentarios introducidos con /* acanban con */ y pueden extenderse varias líneas. Por ejemplo:

```
/*
   Este es un comentario
   que se extiende varias líneas.
*/
console.log('Hola mundo'); /*Introduce 'Hola mundo' en la página*/
/* La siguiente línea no se ejecutará pues está comentada:
 console.log('Adios mundo');
*/
```

4 Toda instrucción acaba en ;

Como se ha visto en los ejemplos anteriores, se ponía un ; al final de cada orden:

```
// Este ejemplo escribe 'Hola mundo Adios mundo'
console.log('Hola mundo');
console.log('Adios mundo'); // Cada orden acaba por ;
```

Hay situaciones en las que si por error no se pone el ; al final de la instrucción o se coloca un ; en el lugar inadecuado, se obtendrán resultados inesperados.

Aunque Javascript permite ignorar los ; al final de cada línea si no hay errores a la hora de interpretar el código:

```
console.log('Hola mundo')
console.log('Adios mundo')
```

5 Variables: Una forma de almacenar datos

Las variables nos proveen una forma de poder almacenar datos en la memoria RAM del ordenador. Van a ser cajas en las que introduciremos nuestros datos. Dentro de una caja podremos introducir varios valores, e incluso otras variables (podemos meter dentro de una caja otras cajas).

5.1 Las tres formas de definir variables

Habrá tres formas de definir las variables:

- Usando *let*
- Usando *const*
- Usando *var*

5.1.1 let

La forma más habitual de definir variables será con let:

```
let a = 1;  // a vale 1
let b = 2;  // b vale 2
let c;      // Se declara c, pero no se le asigna ningún valor
c = a + b;  // Se hace la suma de a + b = 1 + 2 = 3, por lo tanto c = 3
```

Las variables definidas con let, se destruyen al salir del bloque. Un bloque es el código que está encapsulado entre {}.

Por ejemplo:

```
{
 let a = 1;  // a vale 1
 a += 1; // Se le suma 1
} // Al salir del bloque se destruye a
let b = 2;  // b vale 2
let c;      // Se declara c, pero no se le asigna ningún valor
c = a + b;  // Error variable a no definida
```

5.1.2 const

Const se usa para definir referencias a posiciones fijas de memoria. En el caso de asignar un tipo primitivo número, booleano o carácter, no se le puede cambiar el valor.

Por ejemplo:

```
const a = 2;
a = 3; // Error, se intenta cambiar el valor al que apunta una constante
```

Otro ejemplo:

```
const a = {campo1: 1, campo2: 2};
a.campo1 = 3; // Se pueden cambiar los valores de la referencia
console.log(a.campo1);
// Si se descomenta la siguiente línea se produce un error
//a = 3;  // ¡No se puede cambiar el valor al que apunta la referencia!
```

Importante: Se recomienda usar *const* a la hora de definir variables. Javascript *no es un lenguaje fuertemente tipado*, como puede ser Java, y es muy fácil cometer errores a la hora de asignar variables.

Para entender el funcionamiento de const, se puede imaginar que la RAM está dividida en cajas y las variables son flechas que apuntan a esas cajas. Let permite cambiar la caja a la que apunta una flecha. Const no permite cambiar dicha fecha de caja. Puede darse el caso que la misma caja sea apuntada por varias flechas:

```
const a = {campo1: 1, campo2: 2};
let b = a;
const c = a;
b.campo1 = 3;
console.log(c.campo1);
```

Aparece un 3 al ejecutar este código, ya que a, b y c apuntan a la misma caja en memoria.

5.1.3 var

Var se usa para definir variables globales. Cuando una variable se define con var, está disponible en todo el código que se ejecute a partir de esa línea:

```
{
  var a = 1;
} // a no se destruye al salir del bloque
let b = 2;
c = a + b;
```

Importante*: Se desaconseja el uso de var y se recomienda el uso de let.*

5.2 Nombres de las variables

A las variables les podemos poner el nombre que queramos, pero este nombre no debe comenzar por un número y se **recomienda** usar los siguientes caracteres: De la a a la z (minúsculas), de la A a la Z (mayúsculas), del 0 al 9 y el símbolo _.

Son nombres de variables válidos: pepe, Pepe, a, b, b1, b2, hola_mundo, _hola_mundo,...

Son nombres de variables no válidos: 1pepe (comienza por un número), a+a (tiene el símbolo + que no se puede usar), hola-mundo (tiene el signo - que no lo podemos usar),...

Importante: El navegador distingue entre mayúsculas y minúsculas, por lo tanto la variable pepe, será diferente de la variable Pepe y de la variable PePe.

En los ejemplos anteriores se han asignado sólo valores numéricos a las variables, pero también se les pueden asignar otros valores:

- Números ya sean con decimales (coma flotante, float en inglés) o sin decimales (enteros, integer en inglés). Por ejemplo: a = 3
- Cadenas de texto. Por ejemplo: a = "Hola mundo"
- Booleanos: Pueden valer verdadero (true) o falso (falso). Por ejemplo: a = true
- Objetos: Como ya se han comentado son cajas en las que podemos meter dentro otras variables u otras cajas en las que insertar más variables. También pueden contener métodos que son pequeños programas que podemos ejecutar.
- Funciones: Javascript permite crear pequeños programas que podemos ejecutar cuando sea necesario, estos programas reciben el nombre de funciones. Se puede asignar a una variable el código de una función, para hacer que la variable se comporte como una función.

5.3 El operador de asignación

Para asignar un valor a una variable usamos el signo = (operador de asignación):

```
let x = 3;
```

Aparte del operador =, también existen += y -=. El operador += suma un valor a la variable:

```
let x = 3;
x += 5; // Al final x = 8 (suma 5 a 3)
```

Poner x += 5 es equivalente a escribir x = x +5.

5.4 Booleanos

Sólo pueden tomar dos valores verdadero (true) y falso (false).

```
let a = true;
let b = false;
a = 3 > 2;  // a = true, pues 3 es mayor que 2
```

5.5 Números

Las operaciones que podemos hacer con los números son +, -, * (multiplicación) y / (división). Por ejemplo:

```
let a = 3;
let b = 2;
let c;
c = 2 + 3;  // c = 5
c = a - 2;  // c = 1
c = a * 2;  // c = 6
c = a * b;  // c = 6
```

```
c = c / 2;  // c = 3 Recordar que el último valor que había tomado era 6.
```

En el caso de que los números sean enteros, no tengan decimales, se pueden hacer 3 operaciones más:

% Resto de la división. Por ejemplo: a = 5 % 2; // a = 1 (que es el resto de la división de 5 entre 2)

++ Aumenta en uno el valor del número. Por ejemplo:

```
a = 3; a++; // a = 4
```

-- Disminuye en uno el valor del número. Por ejemplo:

```
a = 3; a--; // a = 2
```

Otro ejemplo:

```
a = 5; a++; // a = 6
```

Los operadores ++ y -- tienen una propiedad curiosa que será útil en determinados casos: Cuando van detrás de la variable se incrementa/decrementa el valor de la variable después de habes superado en ;. Cuando van delante de la variable incrementan/decrementan el valor de la variable antes de evaluar ninguna otra variable. Por ejemplo:

```
let a = 2;
let b = 3;
let c;
c = a + b;     // c = 5 Hasta aquí todo normal
c = a + b++ ;   /* ¡ c = 5 ! y b = 4. Antes de llegar el ; b valía 3, por lo tanto
                se hace la operación c = 2 + 3, cuando se supera el ; se hace
                que b incremente su valor */
```

El mismo ejemplo poniendo el operador ++ delante de la variable:

```
let a = 2;
let b = 3;
let c;
c = a + b;     // c = 5 Hasta aquí todo normal
c = a + ++b ;   /* ¡ c = 6 ! y b = 4.  b valía 3 y se incrementa a b = 4,
                por lo tanto se hace la operación c = 2 + 4 */
```

5.6 Cadenas de texto

Javascript permite crear una cadena de texto simplemente poniéndola entre comillas. Por ejemplo:

```
let a = "Hola mundo";
console.log(a);  // Escribirá Hola mundo
```

Para crear una cadena se pueden usar comillas dobles " o simples ':

```
let a = "Hola mundo";
let a = 'Hola mundo';
```

Con comillas simples y dobles sólo podemos introducir textos que ocupen una línea. En el caso de querer introducir textos que ocupen varias líneas se usará el acento invertido ` (es un acento y como los que se ponen a las vocales (àèìòù) pero está invertido). El acento invertido suele estar en el teclado a la derecha de la tecla p. Un ejemplo:

```
let a = `Este texto
   puede ocupar
   varias líneas`
```

La única operación que se puede realizar con las cadenas de texto es la concatenación, que consiste en unir dos cadenas de texto poniéndolas una a continuación de la otra. Se realiza con el operador +:

```
let cadena1 = "Hola";
let cadena2 = "Mundo";
console.log(cadena1+cadena2);   // Escribe HolaMundo
```

5.6.1 Conversión implícita de tipos

Supongamos el siguiente ejemplo:

```
let cadena1 = "Hola";
let b = 2;
console.log(cadena1+b);   // Escribe Hola2
```

En el ejemplo anterior la operación que realiza Javascript es: cadena1 es una cadena de texto, b es un entero. Al realizar cadena1+b convierte b a cadena de texto, por lo que al final tenemos una concatenación de la cadena "Hola" con la cadena "2".

Veamos otro ejemplo:

```
let a = "23";
let b = "45";
console.log(a+b);   // Escribe 2345
```

Como a y b son cadenas de texto, en pantalla aparecerá la cadena "2345", en lugar de hacer la suma de 23+45=68.

Un caso especial son cuando no existe el operador para el tipo dado y el interprete de Javascript improvisa. Por ejemplo, las cadenas de texto poseen el operador + (que concatena), pero no disponen del operador * (multiplicar):

```
let a = "23";
let b = "45";
console.log(a*b);   // Escribe 1035
```

En este caso el intérprete se da cuenta que * no es un operador de cadenas de texto e intenta improvisar. En este caso sí puede realizar la operación de 23*45=1035.

Hay otras situaciones en que esto no es posible:

```
let a = "Hola";
let b = "45";
console.log(a*b);   // Escribe NaN
```

En este caso el intérprete de Javascript escribe NaN (Not a Number) indicando que no se puede realizar la operación.

5.6.2 Conversión explícita de tipos

Puede interesar convertir una cadena de texto a un número. Para ello se usan las funciones:

parserInt(cadena) Convierte una cadena de texto en un número entero (un número sin decimales). Por ejemplo:

```
let a = "23";
let b = 2;
let c = parseInt(a);    // c = 23 número, no cadena de texto
console.log(b+c);    // Escribe 25 la suma de 23 + 2
console.log(a+b);    // Escribe 232, concatena la cadena "23" con 2.
```

parseFloat(cadena) Convierte una cadena de texto en un número en coma flotante (un número con decimales). Por ejemplo:

```
let a = "23.5";
let b = 2;
let c = parseFloat(a);    // c = 23.5 número, no cadena de texto
console.log(b+c);    // Escribe 25.5 la suma de 23.5 + 2
console.log(a+b);    // Escribe 23.52, concatena la cadena "23.5" con 2.
```

También puede interesar convertir un número a cadena de texto. Esto se hará con el método **toString()**:

```
let a = 23.5;
let b = 2;
console.log(a.toString()+b.toString());    // Escribe 23.52, concatena la cadena "23.5" con "2".
```

5.7 undefined y null

Una variable declarada a la que no se le ha asignado ningún valor, se le asigna por defecto el valor **undefined**:

```
let a;
console.log(a);  // Se escribe undefined, pues a a no se le ha asignado ningún valor
```

null se usa cuando se desea borrar de la RAM alguna variable. Por ejemplo:

```
let a = 2;  // Se alamacena un 2 en la RAM
a = null;   // Se libera el espacio asignado al 2 de la RAM
```

6 Estructuras de control de flujo

6.1 if

Con la estructura if se puede controlar si se ejecuta una determinada parte del código. Por ejemplo, se le puede pedir al usuario su edad y si es menor de edad, no se le permita el acceso.

La estructura if tiene la siguiente forma:

```
if(codición) {
   // Se ejecuta si la condición es cierta
}
```

Por ejemplo:

```
let a = 2;
let b = 3;
if( a < b) {
   console.log("a es menor que b");
```

```
}
```

En el ejemplo anterior a vale 2 y b vale 3. La condición dice que como a es menor que b (2 < 3) entonces se escribirá "a es menor que b".

Otro ejemplo:

```
let a = 2;
let b = 3;
if( a > b) {
   console.log("a es mayor que b");
}
```

Es este caso como a es menor que b, no se llega a entrar dentro del if.

Las llaves después del if { ... }, se colocan cuando se van a ejecutar varias instrucciones. En el caso de que sólo se vaya a ejecutar una, la estructura se puede simplificar quitando las llaves:

```
let a = 2;
let b = 3;
if( a > b)
   console.log("a es mayor que b");
```

6.1.1 Operadores de comparación

Según el tipo de datos que se esté comparando, se pueden usar los siguientes comparadores. Por ejemplo, con números podremos usar los comparadores:

- > Mayor que por ejemplo 3 > 2
- < Menor que por ejemplo 2 < 3
- >= Mayor o igual que por ejemplo 3 >= 2
- <= Menor o igual que por ejemplo 2 <= 3
- == Igual a por ejemplo 3 == 3
- != Distinto a por ejemplo 4 != 3

Con cadenas de texto y booleanos se pueden usar los comparadores:

- == Igual a por ejemplo "Hola" == "Hola"
- != Distinto a por ejemplo "Hola" != "adios"

En el caso de comparar cualquier variable, se pueden usar los comparadores == y !=.

6.2 if - else

Un ejemplo de comparación de cadenas:

```
let entrada = “12345”;
if( entrada == '12345') {
   console.log('Acceso concedido');
}
```

En el ejemplo anterior se solocita una contraseña al usuario, si es 12345, aparece el mensaje acceso concedido. Pero supongamos que cuando el usuario teclee una contraseña incorrecta, se escriba "Contraseña incorrecta". Para estos casos se usa la estructura else:

```
if(condición) {
   // Se ejecuta si la condición es cierta
} else {
   // Se ejecuta si la condición es falsa
}
```

Así el ejemplo anterior quedaría:

```
let entrada = “12345”;
if( entrada == '12345') {
   console.log('Acceso concedido');
} else {
      console.log('Acceso denegado');
}
```

6.3 else if

Hay situaciones en las que se pueden dar varias condiciones. Por ejemplo, imaginemos una taquilla de un cine que en función de la edad hagan descuentos diferentes. Se tendrán entonces varias condiciones en función de la edad como podrían ser los menores de 10 años pagan 5 €, los menores de 18 € pagan 6 €, los menores de 60 años pagan 10 € y los mayores de 60 no pagan.

Para estas situaciones se usa la estructura if - else if - else:

```
if(condición 1) {
   // Se ejecuta si se cumple la condición 1
} else if (condición 2) {
   // Se ejecuta si se cumple  la condición 2
} else if (condición 3) {
   // Se ejecuta si se cumple  la condición 3
}  else if (condición n) {
   // Se ejecuta si se cumple  la condición n
} else {
   // Se ejecuta si ninguna de las condiciones
   // se cumple.
}
```

Por ejemplo:

```
let edad = 10;
let precio;
if(edad < 10) {
   precio = 5;
} else if(edad < 18) {
   precio = 6;
} else if(edad < 60) {
   precio = 10;
} else {
   precio = 0;
}
```

Como en el caso anterior edad = 10 el precio será 6 €.

El orden de las condiciones es importante, por ejemplo:

```
let edad = 18;
let precio;
```

```
if(edad < 60) {
   precio = 10;
} else if(edad < 10) {
   precio = 5;
} else if(edad < 18) {
   precio = 6;
} else {
   precio = 0;
}
```

Como la primera condición es edad < 60 se entra en la primera condición, precio = 10 y no se siguen evaluando las siguientes condiciones.

6.4 Operaciones lógicas

En muchos casos se van a necesitar condiciones del tipo "si esto y esto otro es verdad entonces ...", es decir, que se cumplan varias condiciones a la vez o al menos una condición entre varias. Para esto se usan las operaciones lógicas.

Las operaciones lógicas más comunes son:

- && Operador y. Es cierto cuando todas las condiciones son ciertas. Por ejemplo:

```
let x = 10;
if ( x < 20 && x > 5 ) {
   // Se ejecuta cuando x es menor que 20 y x mayor que 5.
   console.log('El número está entre 5 y 20');
}
```

Si es necesario se pueden poner varias condiciones:

```
let x = 10;
let b = 3;
if ( x < 20 && x > 5 && b == 3 ) {
   // Se ejecuta cuando x es menor que 20 y x mayor que 5
   // y b es igual a 3.
   console.log('El número está entre 5 y 20 y b es igual a 3');
}
```

- || Operador o. Es cierto cuando alguna de las condiciones es cierta. Por ejemplo:

```
let x = 10;
let b = 3;
if ( x < 20 || x > 5 || b == 3 ) {
   // Se ejecuta cuando x es menor que 20 o x mayor que 5
   // o b es igual a 3.
   console.log('Se cumple una de las condiciones');
}
```

- ! Operador de negación. Si la condición es cierta devuelve falso, si es falsa, devuelve cierto:

```
let x = 11;
if( ! x == 10 ) {
   // Se ejecuta cuando x no es igual a 10.
   // Es equivalente a x != 10
```

```
    ...
}
```

A veces va a interesar mezclar operadores. Por ejemplo:

```
let x = parseInt( prompt('Escribe un número del uno al diez') );
if ( (x >= 1 && x <= 10) && (x == 2 || x == 3 || x == 5 || x == 7) ) {
    console.log('El número es un número de 1 a 10 y es primo');
}
```

Como se puede ver en el ejemplo, se usan paréntesis para agrupar las condiciones. Por un lado se agrupa la condición (x >= 1 && x <= 10) y por otro las condiciones de si es número primo.

A parte de los operadores && y || existen los operadores & y |. Su uso es igual que el de && y || respectivamente, pero existe una diferencia. Fíjemonos en el siguiente código:

```
let x = parseInt( prompt('Escribe un número del uno al diez') );
if ( x == 2 | x == 3 | x == 5 | x == 7) {
    console.log('El número es un número primo');
}
if ( x == 2 || x == 3 || x == 5 || x == 7) {
    console.log('El número es un número primo');
}
```

Imaginemos que x = 3. En la condición (x == 2 | x == 3 | x == 5 | x == 7) se evalua x==2 y se ve que es falsa, se evalua x==3 y se ve que es cierta, se evalua x==5 y se ve que es falsa y finalmente se evalua x==7 y se ve que es falsa. Como una de las condiciones es cierta (x==3) la condición es cierta.

Imaginemos que x = 3. En la condición (x == 2 || x == 3 || x == 5 || x == 7) se evalua x==2 y se ve que es falsa, se evalua x==3 y se ve que es cierta. Como el operador || es cierto si alguna de las condiciones es cierta, no es necesario seguir evaluando.

Se pueden ejecutar funciones dentro de las condiciones, con los operadores & y | nos aseguramos que esas funciones se ejecuten:

```
let x = 3;
let y = 7;
if ( x == 2 | x == 3 | y == prompt('Escribe un número del uno al diez')) {
    console.log('El x es 2 ó 3 e y vale 7');
}
if ( x == 2 || x == 3 || y == prompt('Escribe un número del uno al diez')) {
    console.log('Si x es 2 ó 3 no nos llega a preguntar el número');
}
```

En el ejemplo anterior el primer if siempre ejecutará la función prompt. En el segundo, si x vale 2 ó 3, no lo preguntará.

6.5 Bucle while

El bucle while repite un código mientras la condición sea cierta. Su estructura es:

```
while(condición) {
```

```
    // Esta parte se repite mientras la condición sea cierta
    ...
}
```

Por ejemplo:

```
let x = 0;
while(x<10) {
    console.log(x);
    x++;
}
```

Este código escribe los números del 0 al 9.

El bucle while puede ayudar a escribir menos código dentro de una página web. Por ejemplo, el siguiente código escribe la tabla de multiplicar del 4:

```
let numero = 4;
let x = 0;
while(x<=10) {
    console.log(numero + 'x' + x + ' = ' + (x*numero));
    console.log('<br/>');
    x++;
}
```

Todo bucle tiene que cumplir lo siguiente: Tiene que tener algo que haga parar el bucle, en el caso anterior la condición (x<10) y algo que haga avanzar al bucle (x++). En caso contrario el bucle se ejecutará hasta el infinito. En estos casos el programa puede dejar de responder y dar la sensación de que "se ha colgado".

break es una forma de forzar la salida de un bucle. Por ejemplo:

```
let numero = 4, x = 0;
while(true) {
    console.log(numero + 'x' + x + ' = ' + (x*numero) + '<br/>');
    x++;
    if(x == 11) break;
}
```

En este caso la condición del while es siempre (true), por lo que el bucle se ejecutaría infinitamente. Pero más adelante se tiene la condición:

```
if(x == 11) break;
```

que hace que cuando x sea 11, el bucle se pare.

continue se usa para pasar a la siguiente iteración del bucle ignorando las instrucciones que haya a continuación. Por ejemplo:

```
let x = 0;
while(x<10) {
    console.log(x + '<br/>');
    x++;
    continue;
    // Estas líneas no se llegan a ejecutar nunca:
```

```
    console.log('Hola mundo');
}
```

En el siguiente ejemplo se puede ver un uso más claro de while: Se escriben los números del 1 al 10 salvo el 3 y el 4:

```
let x = 0;
while(x<10) {
    x++;
    if(x==3 || x==4) {
        continue;
    }
    console.log(x + '<br/>');
}
```

6.6 Bucle for

Un bucle muy utilizado es el bucle for. Con el bucle while se pueden hacer todas las operaciones de los bucles, pero el bucle for es una forma de while que se usa mucho. Lo emplearemos cuando que haya que hacer una enumeración (por ejemplo, contar del 1 al 100). La estructura del bucle for es la siguiente:

```
for(asignación-inicial; condición; avance) {
    // Instrucciones a repetir
}
```

La asignación inicial sólo se ejecuta una vez antes de iniciar el bucle, la condición y el avance se evaluan a cada paso del bucle. El bucle finaliza cuando acaba la condición. Es equivalente al siguiente bucle while:

```
asignacion-inicial;
while(condición) {
    // Instrucciones a repetir
    avance;
}
```

Un ejemplo contando los números del 0 al 9:

```
let x;
for(x=0; x<10; x++) {
    console.log(x);
}
```

Al igual que en el bucle while, se puede usar break para parar la ejecución del bucle:

```
let x;
for(x=0; x<100; x += 2) {
    console.log(x);
    if(x==10) break;
}
```

El código anterior escribe los números pares del 0 al 10.

6.7 Bucle do - while

Es un bucle muy similar al bucle while, pero las instrucciones a repetir se ejecutan al menos una vez. Su estructura:

```
do {
   // Instrucciones a repetir
} while(condición);
```

Por ejemplo, para escribir los números del 0 al 9:

```
let x = 0;
do {
   console.log(x);
   x++;
} while(x<10);
```

6.8 switch - case

Existen ocasiones en los que hay que usar condiciones como las siguientes:

```
let x = 10;
if( x == 2) {
   console.log('Pague dos y lleve una');
} else if( x == 3) {
   console.log('Pague tres y lleve dos');
} else if( x == 4) {
   console.log('Pague cuatro y lleve dos');
} else {
   console.log('En esta tienda siempre te engañan');
}
```

En el caso en el que se tengan varias condiciones en las que un número entero deba ser igual a otro entero, se puede simplificar usando las estructura switch - case. Sólo vale para números enteros:

```
let x = 10;
switch( x ) {
   case 2:
      console.log('Pague dos y lleve una');
      break;
   case 3:
      console.log('Pague tres y lleve dos');
      break;
   case 4:
      console.log('Pague cuatro y lleve dos');
      break;
   default:
      console.log('En esta tienda siempre te engañan');
}
```

Su sintaxis es:

```
switch( variable ) {
   case valor-1:
      // Se ejecutan si variable == valor-1
      break;
   case valor-2:
      // Se ejecutan si variable == valor-2
```

```
        break;
    // ...
    case valor-n:
        // Se ejecutan si variable == valor-n
        break;
    default:
        // Se ejecuta si no se cumple ninguna de las condiciones anteriores
}
```

6.9 Estructuras anidadas

Dentro de una estructura se van a poder introducir otras, un if dentro de un while, un while dentro de un if, un if dentro de otro if,...

Mención a parte merecen los bucles. Por ejemplo:

```
let x, y;
for(x=0; x<10; x++) {
    for(y=0;y<10;y++) {
        console.log(x * y);
        console.log(':');
    }
    console.log('<br/>');
}
```

En el ejemplo anterior hay un bucle for dentro de otro bucle for. El bucle for de la variable x se ejecuta 10 veces (del 0 al 9). El bucle for de la variable y cuenta del 0 al 9 y está dentro del bucle for de la x, por lo tanto el console.log(x*y) se va a repetir 100 veces (10 veces del bucle for de x y 10 veces del bucle for de y).

7 Funciones

En muchas ocasiones la misma porción de código es necesaria ejecutarla en varias partes. Por ejemplo, tenemos varias páginas web y todas tienen la misma barra de navegación. Podríamos copiar y pegar el código de la barra de navegación en cada página. Pero si hacemos una modificación en la barra de navegación tendríamos que ir página por página añadiendo dicha modificación.

Una forma muy cómoda de trabajar con partes de código que se van a repetir mucho son las funciones. La sintaxis de una función es la siguiente:

```
function nombre_de_la_función(argumento1, argumento2, argumento3, ...) {
    // Código a ejecutar
    return valor_a_devolver;
}
```

Por ejemplo, supongamos una función que escribe un texto:

```
function texto() {
    console.log("Este es un texto largisimo");
    console.log("Más texto");
}
```

Para ejecutarlo, simplemente llamaríamos a la función por su nombre:

```
texto();
```

Es decir, cade vez que se escriba "texto()", será como si se sustituyera por el código de la función.

Por ejemplo:

```
function texto() {
   console.log("Este es un texto largisimo");
   console.log("Más texto");
}

// Se llama a la función cuando es necesario
texto();
// Se llama a la función cuando es necesario
texto();
```

En el ejemplo anterior se puede ver la ventaja de las funciones, si modifico la función se modifica la salida de la función en todos los puntos en donde se la invoque.

7.1 Funciones con argumentos

El ejemplo anterior era una función que no admitía argumentos. Se le pueden pasar argumentos a la función de forma que estos son pasados como variables que la función puede manejar. Por ejemplo:

```
function suma(a,b) {
   console.log(a+b);
}
```

A la función anterior se le van a pasar dos argumentos a y b. La función escribirá el resultado de sumar ambos valores. Es decir, cuando se escriba:

```
suma(2, 3);
```

Esto será equivalente a:

```
{
   var a = 2;
   var b = 3;
   console.log(a+b);
}
```

Se pueden buscar ejemplos más elaborados, como el siguiente que muestra las tablas de multiplicar:

```
// Esta función escribe la tabla de multiplicar del número que se le pase:
function tabla(numero) {
   var x;
   for(x=0;x<=10;x++) {
      console.log(numero + 'x' + x + '=' + (numero*x) + '\n');
   }
   console.log('\n');
}

// Se muestan las tablas de multiplicar del 0 al 10
let n;
for(n = 0; n <= 10; n++)
   tabla(n);
```

En el ejemplo anterior no es necesario el uso de funciones, pero hacen que ciertas partes del código resulten más legibles.

7.2 Funciones que devuelven valores

Para la función devuelva un valor se usa return. Veamos el siguiente ejemplo:

```
// Calcula el precio más un 21 % de IVA
function iva(precio) {
   return precio*1.21;
}
```

Esta función luego se puede llamar donde sea necesaria y recoger el valor devuelto. Por ejemplo:

```
// Calcula el precio más un 21 % de IVA
function iva(precio) {
   return precio*1.21;
}

let a_pagar = 0;
a_pagar += iva(100);
a_pagar += iva(20);

console.log("El total a pagar es: " + a_pagar);
```

En el siguiente ejemplo, la función calcula la suma de los números de a a b:

```
// Esta función calcula la suma de los números de a a b:
function suma_numeros(a,b) {
   let x;
   let suma = 0;
   for(x=a;x<=b;x++) {
      suma += x;
   }
   return suma;
}

console.log("La suma de los números del 1 al 100 es: "+ suma_numeros(1,100) );
console.log("La suma de los números del 23 al 58 es: "+ suma_numeros(23,58) );
```

Una función puede tener varios return, pero cuando se llegue a ejecutar el primero, el programa sale de la función y no continua evaluándola:

```
// Esta función calcula la suma de los números de a a b:
function suma_numeros(a,b) {
   let x;
   let suma = 0;
   for(x=a;x<=b;x++) {
      suma += x;
   }
```

```
    return suma;

    // El resto de la función no se continua evaluando:
    console.log("Hola mundo"); // Esto no llega a escribirse.
}

console.log("La suma de los números del 1 al 100 es: "+ suma_numeros(1,100) );
```

return se puede usar dentro de la estructura if - else para devolver diferentes respuestas:

```
// Esta función devuelve el precio a pagar en función de la edad
function precio(edad) {
    if(edad < 10)
        return 5;
    else if(edad < 18)
        return 10;
    else if(edad < 60)
        return 15;
    else
        return 5;
}

var edad_cliente = 18;
console.log("Debe pagar: "+ precio(edad_cliente) );
```

Se puede usar return sin argumentos para salir de una función que no devuelve valores. Por ejemplo:

```
// Esta función devuelve el precio a pagar en función de la edad
function precio(edad) {
    if(edad < 10) {
        console.log("Debe pagar: 5");
        return ;  // Return no tiene argumentos, no devuelve nada
    } else if(edad < 18) {
        console.log("Debe pagar: 10");
        return ;
    } else if(edad < 60) {
        console.log("Debe pagar: 15");
        return ;
    } else
        console.log("Debe pagar: 5");
}

let edad_cliente = 18;
precio(edad_cliente);
```

7.3 Funciones como variables

Una característica útil es la de poder almacenar a las funciones como si fueran variables, e incluso pasarlas como argumentos de funciones:

```
  let iva = function(precio) {
     return precio*1.21;
  }

  let b = iva;

  console.log(iva(10));
  console.log(b(20));
</script>
```

Como se puede ver se asigna a la variable iva una función (esta función no tiene nombre, por lo que se denomina **función sin nombre o función lambda**). Luego se asigna iva a la variable b. Por lo que tanto la función iva como la función b hacen lo mismo.

Puesto que actúan como variables se pueden pasar como argumentos de otras funciones. Por ejemplo:

```
  let iva = function(precio) {
     return precio*1.21;
  }

  function calcular(precio, operacion)
  {
     precio += operacion(10);
     return precio;
  }

  let resultado = calcular(10, iva); // Se le pasa como argumento la función iva
  console.log(resultado);
```

Se puede, incluso, definir la función cuando se pasa como argumento de otra función:

```
  function calcular(precio, operación) {
     precio += operacion(10);
     return precio;
  }

  // Se le pasa como argumento la función sin nombre que calcula en iva
  let resultado = calcular(10,  function(precio) {return precio*1.21;} );
  console.log(resultado);
```

Otra forma de definir funciones como variables es con la siguiente sintaxis:

(argumentos) => {código de la función}

Por ejemplo, se ha definido la función calcularIVA en el siguiente ejemplo:

```
let calcularIVA = (precio) => {
   var iva = precio * 1.21;
   return iva;
}

// Se le pasa como argumento la función sin nombre que calcula en iva
let resultado = calcularIVA(100);
console.log(resultado);
```

Las funciones como variables se suelen usar mucho para tratar eventos o definir objetos en Javascript.

8 Objetos

Uno objeto será como una caja en la que vamos a poder almacenar otras cajas. Es decir, tendremos una variable y dentro de esta variable se podrán almacenar más variables o funciones. Por ejemplo, se ha estado usando la función "console.log()". Lo que realmente se ha estado haciendo es del objeto **console**, usar la variable **log** que almacena una función. El operador "." (punto) sirve para obtener los contenidos de un objeto. Por ejemplo:

```
// Se almacena dentro del objeto document la variable a:
console.a = 10;
// Se almacena dentro del objeto document la variable b:
console.b = 20;
// Se hacen operaciones con las variables:
let c = console.a + console.b;
console.log(c);
```

Como se ve en el ejemplo, dentro del objeto console, se han almacenado las variables a y b y nos referíamos a ellas como console.a y console.b.

También se puede ver que se puede modificar cualquier objeto que llegue a nuestro código. En otros lenguajes, como Java, no se pueden añadir atributos o métodos a un objeto ya definido.

8.1 Definiendo objetos a partir funciones

Hay varias formas de definir un objeto en Javascript. Lo más común es usar una función. La palabra clave **this** se referirá al objeto que estamos creando o manejando. Por ejemplo:

```
function Persona (edad) {
   this.edad = edad;
   this.nombre = "Pepe";
```

```
    this.altura = 1.70;
    this.peso = 70;
    this.engordar = function (pesoGanado) {
        this.peso += pesoGanado;
    }
}

let pepe = new Persona(20);
console.log("Edad " + pepe.edad);
console.log("nombre " + pepe.nombre);
console.log("altura " + pepe.altura + "m”);
console.log("peso " + pepe.peso + "kg");
pepe.engordar(11);
console.log("Ha engordado. peso " + pepe.peso);
```

Como se puede ver dentro de la función Persona se hacen varias referencias a **this** que es el objeto que se está creando. Se puede apreciar que se le han asignado las variables edad, altura y peso. También se le ha agregado la función engordar.

A las variables que se definen dentro de un objeto se las llama **atributos**. A las funciones que se definen dentro de un objeto se las llama **métodos**.

El ejemplo anterior se puede ver que para crear un nuevo objeto se usa el operador **new**. Usando **new** se crea el objeto pepe y se acceden a sus atributos y métodos:

```
let pepe = new Persona(20);
console.log("Edad " + pepe.edad + "<br/>");
console.log("nombre " + pepe.nombre + "<br/>");
console.log("altura " + pepe.altura + "m <br/>");
console.log("peso " + pepe.peso + "kg <br/>");
pepe.engordar(11);
console.log("Ha engordado. peso " + pepe.peso);
```

8.2 Referencias a objetos

Las variables tienen una forma especial de tratar a los objetos. Veamos el siguiente ejemplo:

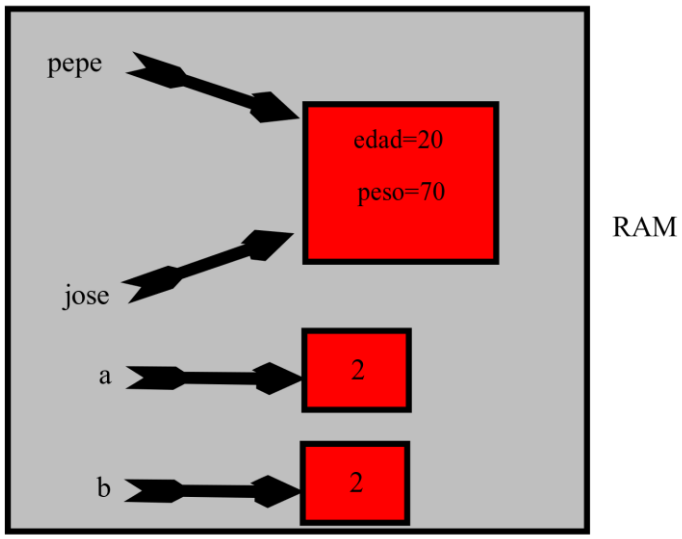
```
function Persona (edad) {
    this.edad = edad;
    this.nombre = "Pepe";
    this.altura = 1.70;
    this.peso = 70;
    this.engordar = function (pesoGanado) {
        this.peso += pesoGanado;
    }
}

let pepe = new Persona(20);
let jose = pepe;
jose.engordar(10);
console.log(pepe.peso);

let a = 2;
let b = a;
```

```
b += 2;
console.log('a='+a);
```

Por defecto cuando se crea el objeto pepe, su peso es 70. Hemos asignado a la variable jose, la variable pepe. Después se ejecuta jose.engordar(10). Cuando se muestra el peso de pepe, se puede ver que pepe ha engordado. ¿Por qué? Porque jose y pepe son el mismo objeto. En la memoria RAM del equipo se crea una zona para almacenar al objeto pepe, cuando se hace jose=pepe, se le está diciendo a la variable jose que apunte a donde está almacenado el objeto pepe:



Pero para el caso de variables que no son objetos, como los números, javascript sí saca una copia, como se puede ver en las variables a y b. Al hacer b=a, javascript saca una copia de a y se la asigna a b por lo que a y b apuntan a regiones diferentes de la RAM. Por ello, al modificar b, no se modifica a.

En general se sacarán copias de los tipos primitivos, en lugar de crear objetos.

9 Estructuras de datos

Los datos se pueden agrupar para formar diversas estructuras. Javascript da herramientas para trabajar con Arrays, listas, pilas, colas, mapas,...

9.1 Arrays

Un array son un conjunto de variables a las que podemos acceder usando un número que le hemos asignado a cada una. Por ejemplo:

```
var a = new Array();
a[0] = 1;
a[1] = 2;
a[2] = a[1] + a[2];
```

Como se puede ver en el ejemplo, se puede declarar un array con "new Array()". Después podemos asignar valor usando la sintaxis:

```
variable_array[índice]
```

Así se escribe a[0] para poder acceder al primer elemento del array, a[1] al segundo elemento,...

En Javascript los arrays se comienzan a numerar desde 0.

Para saber cuántos elementos hay en el array se usa la propiedad length:

```
let a = new Array();
a[0] = 1;
a[1] = 2;
a[2] = a[1] + a[2];
alert(a.length); // Escribe 3, pues se han añadido los elementos a[0], a[1] y a[2]
```

Se suelen usar bucles para recorrer los contenidos de los arrays:

```
let a = new Array();
let i;
a[0] = 1;
a[1] = 2;
for(i=2;i<100;i++)
   a[i] = a[i-1] + a[i-2];
// En el array se han almacenado la suma de los valores anteriores:
for(i=0;i<a.length;i++) {
   console.log(a[i] + '<br/>');
}
```

9.1.1 Arrays multidimensionales

Los ejemplos anteriores trabajaban con arrays en una dimensión. Pero se puede insertar un array dentro de otro y se manejarán arrays en varias dimensiones:

```
let i, j, a = new Array();
// Se crea un array con 10 elementos que serán arrays
for(i=0;i<10;i++)
   a[i] = new Array();
// Se puede acceder a cada elemento de cada subarray:
// El a[0] será el primer array, a[0][0] será el primer elemento del primer array:
a[0][0] = 1;
a[0][1] = 2;
// Se usarán bucles anidados para poder recorrerlos:
for(i=0;i<10;i++) {
   for(j=0;j<10;j++) {
      a[i][j] = i*j;
   }
}
console.log(a);
```

En el ejemplo anterior se ha definido un array de dos dimensiones (necesitamos dos índices para recorrerlo). Se pueden definir arrays de 3, 4 ó más dimensiones.

9.1.2 Otra forma de definir arrays

Hasta ahora se han definido los arrays usando "new Array()". Se pueden definir arrays usando otras notaciones. Otra forma muy usada es utilizar corchetes [...]. Por ejemplo:

```
let i;
let lista = ['Huevos', 'Pan', 'Leche', 'Frutas', 'Patatas'];
console.log('La lista de la compra tiene ' + lista.length + ' elementos: <br/>');
for(i=0;i<lista.length;i++)
   console.log(lista[i]);
```

Como se puede ver en el ejemplo, sólo hay que poner los elementos del array entre corchetes y separados por comas:

```
[elemento0, elemento1, elemento2,...]
```

Dentro de los corchetes se pueden introducir elementos de cualquier tipo:

```
let i;
let lista = [1, 'Pan', new Array() ];
lista[2] = [1,2,3];
console.log('La lista tiene ' + lista.length + ' elementos: <br/>');
for(i=0;i<lista.length;i++)
   console.log(lista[i]);
```

Se pueden introducir incluso otros arrays como se puede ver en el ejemplo anterior. Se pueden definir arrays multidimensionales de la siguiente forma:

```
let i, j;
// Se define el array lista con 3 arrrays y cada array
// tiene 3 elementos:
let lista =
   [
      [1,2,3],
      [4,5,6],
      [7,8,9]
   ];
// Se recorren los elementos de lista:
for(i=0;i<lista.length;i++) {
   for(j=0;j<lista[i].length;j++) {
      console.log(lista[i][j] + ' ');
   }
   console.log('<br/>');
}
```

9.1.3 Métodos del objeto Array

En el siguiente enlace se pueden encontrar diversos métodos que se pueden usar con un array:

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array

Entre los más útiles están sort() que ordena un array, indexOf(valor) que devuelve el índice del primer elemento que sea iguan a valor,...

```
let i;
// Este array está desordenado:
let lista = [8,9,5,6,3,4,7,2,1];
// Se ordena:
lista.sort();
// Se muestra ordenado:
for(i=0;i<lista.length;i++) {
	console.log(lista[i] + ' ');
}
```

9.2 Colas

Con los arrays se pueden simular otras muchas estructuras de datos. Un ejemplo típico es la cola. Imagina por un instante la cola del cine el primero de la cola es el que entra en el cine, los que llegan después se tienen que poner al final de la cola. Para ello se usarán el método **shift()** que extrae el primer elemento y **push(valor**) que añade el valor al final de la cola. Por ejemplo:

```
let cola = [8,9,5,6];
console.log(cola);
// Se extrae el primer elemento de la cola
let primero = cola.shift();
console.log(primero);
// A la cola le falta el primer elemento
console.log(cola);
//Se añade un elemento:
cola.push(7);
console.log(cola);
```

9.3 Pilas

Otra estructura que se usa con frecuencia en la pila. Se puede simular como una pila de platos en los que se ponen unos encima de los otros. Para poder acceder al plato del fondo de la pila, hay que quitar, uno a uno, los platos que tiene encima. Para ello se usarán el método **pop()** que extrae el último elemento y **push(valor**) que añade el valor al final de la pila.

```
let pila = [8,9,5,6];
console.log(pila);
// Se extrae el último elemento de la pila
let ultimo = pila.pop();
```

```
console.log(ultimo);
// A la pila le falta el último elemento
console.log(pila);
//Se añade un elemento:
pila.push(7);
console.log(pila);
```

9.4 Mapas

Un mapa consiste en buscar una variable por una etiqueta que se le ha asociado. La idea es similar al array, pero en lugar de asignar números a cada elemento, se le asigna una etiqueta. Para definir un mapa se igualará a unas llaves {}:

```
let mapa = {};
```

Para asignar una etiqueta a un valor se usa una sintaxis muy similar a la de los arrays:

```
mapa['etiqueta'] = valor;
```

Por ejemplo:

```
let telefonos = {};
telefonos['pepe'] = 555123;
telefonos['juan'] = 555333;
console.log(' El teléfono de pepe es ' + telefonos['pepe']);
console.log(' El teléfono de juan es ' + telefonos['juan']);
```

Otro ejemplo en el que se pueden consultar los teléfonos de varias personas:

```
let telefonos = {};
telefonos['pepe'] = 555123;
telefonos['juan'] = 555333;
var nombre = ‘pepe’;
console.log(' El teléfono de '+ nombre +' es ' + telefonos[nombre]);
```

En el caso de que tengamos que definir un mapa, una forma rápida es:

```
var mapa = {
   'etiqueta1' : valor1,
   'etiqueta2' : valor2,
   'etiquetan' : valorn
};
```

Por ejemplo:

```
let telefonos = {
                'pepe': 555123,
```

```
            'juan': 555333
          };
let nombre = 'juan';
console.log(' El teléfono de '+ nombre +' es ' + telefonos[nombre]);
```

9.5 Usando mapas para definir objetos

Esta forma de definir mapas, puede servir para crear objetos. Se puede acceder a los elementos de la misma forma que se accede a los métodos o propiedades de los objetos:

```
mapa = {
   'etiqueta1' : valor1,
   'etiqueta2' : valor2,
   'etiquetan' : valorn
};
mapa.etiqueta1 = valor;
```

Por ejemplo:

```
let telefonos = {
            'pepe': 555123,
            'juan': 555333
          };
console.log(' El teléfono de pepe es ' + telefonos.pepe);
```

Incluso se pueden definir métodos en los objetos:

```
let contador = {
            n : 0,
            incrementar : function() {
               this.n += 1;
            },
            decrementar : function() {
               this.n -= 1;
            }
         };
console.log('Valor inicial de contador.n: ' + contador.n);
contador.incrementar();
console.log('Se ejecuta contador.incrementar(),  contador.n: ' + contador.n);
contador.decrementar();
console.log('Se ejecuta contador.decrementar(),  contador.n: ' + contador.n);
```

En el ejemplo anterior se puede apreciar que cuando se crean objetos usando mapas, se pueden eliminar las comillas de las etiquetas:

```
mapa = {
   etiqueta1 : valor1,
   etiqueta2 : valor2,
```

```
    etiquetan : valorn
};
mapa.etiqueta1 = valor;
```

10 Node.js

Javascript tiene una fuerte presencia en los navegadores. A alguien se le ocurrió la idea de tomar el motor de Javascript del navegador web Chromium y hacer de él un lenguaje que se pudiese ejecutar localmente. De ahí se origina "Node.js" (se suele quitar el punto cuando se escribe el nombre del proyecto Nodejs).

Ya hubo intentos de hacer de Javascript un lenguale del lado del servidor, como Rhino, pero es "Node.js" el que mñas éxito ha tenido, contando con un repositorio de bibliotecas de lo más extensa.

10.1 Instalación

Para instalar Node.js simplemente se va a la página web de descarga de Node.js y de descarga la versión correspondiente a nuestro sistema operativo:

https://nodejs.org

Para instalarlo en sistemas Linux - Unix no es necesario tener permisos de usuario root. Simplemente se descarga y se descomprime el proyecto en la carpeta que se desee. Generará un árbol de directorios similar a:

- /bin
- /include
- /lib
- /share

Para poder ejecutar "Node.js" sin problemas, es conveniente introducirlo en la variable de entorno $PATH de nuestro sistema operativo. En Linux - Unix, esto se puede realizar modificando el archivo oculto ".bashrc". Para ello en un terminal se ejecuta:

```
$ nano .bashrc
```

Al final del archivo se añade la siguiente línea.

```
export PATH=$PATH:ruta a la carpeta de "Node.js"/bin
export NODE_PATH=ruta a la carpeta de "Node.js"/lib/node_modules
```

10.2 El intérprete de comandos

Al ser concebido como un lenguaje de script, se dispone de un intérprete de comandos. Sólo hay que ejecutar en un terminal el comando:

```
$ node
```

Para salir del intérprete se escribirá:

```
.exit
```

Desde este intérprete se tiene la opción de autocompletar usando el tabulador. Por ejemplo, se crea un objeto String:

```
Welcome to Node.js v12.18.3.
Type ".help" for more information.
> var a= new String();
undefined
```

Al ejecutar "var a= new String();", "Node.js" responde "undefined", pues este comando no devuelve una respuesta que "Node.js" pueda mostrar por el terminal.

Si se escribe "a." y se pulsa el tabulador un par de veces, nos mostrará los métodos y propiedades del objeto a:

```
> a.
a.__defineGetter__      a.__defineSetter__      a.__lookupGetter__      a.__lookupSetter__      a.__proto__
a.hasOwnProperty
a.isPrototypeOf         a.propertyIsEnumerable  a.toLocaleString

a.anchor                a.big                   a.blink                 a.bold                  a.charAt                a.charCodeAt
a.codePointAt           a.concat                a.constructor           a.endsWith              a.fixed                 a.fontcolor
a.fontsize              a.includes              a.indexOf               a.italics               a.lastIndexOf           a.link
a.localeCompare         a.match                 a.matchAll              a.normalize             a.padEnd                a.padStart
a.repeat                a.replace               a.search                a.slice                 a.small                 a.split
a.startsWith            a.strike                a.sub                   a.substr                a.substring             a.sup
a.toLocaleLowerCase     a.toLocaleUpperCase     a.toLowerCase           a.toString              a.toUpperCase           a.trim
a.trimEnd               a.trimLeft              a.trimRight             a.trimStart             a.valueOf

a.length
```

Esto será útil para explorar los contenidos de un objeto dado.

10.3 Ejecutando un archivo

Lo más habitual, para trabajar con "Node.js", será escribir un archivo con los comandos que se necesiten y después ejecutarlo desde línea de comandos con:

```
$ node archivo.js
```

Por ejemplo, si se escribe en el archivo "hola.js":

```
console.log('Hola mundo');
```

y se ejecuta, se escribirá en el terminal:

```
Hola mundo
```

El método "console.log()" se usará para escribir en la salida estándar.

10.4 require

En "Node.js" se dispone de "require", que es similar el *import* de Java o al *#include* de C. Permite cargar los métodos y variables que se exporten desde un archivo. A los archivos que tengan contenidos que se puedan importar desde otros, se llamarán módulos. Por ejemplo, se tiene el siguiente archivo llamado "mates.js":

```
// Archivo mates.js
```

```
// Se definen algunas funciones y valores
let PI = 3.14159

function suma(a, b) {
   return a + b;
}

function resta(a, b) {
   return a - b;
}

// Se indica los elementos que se desean exportar:
exports.PI = PI;
exports.suma = suma;
exports.resta = resta;
```

Con exports se indican los métodos y variables que se desean exportar.

Ahora si se inicia una sesión del intérprete de comandos:

```
$ node
```

y se ejecuta el comando require de la siguiente forma:

```
const mates = require('./mates');
```

Se habrá importado el módulo "mates.js". Hay que fijarse que en el argumento de require hay que indicar la ruta al archivo sin la extensión. Lo habitual es crear una constante, const, con el módulo cargado. El nombre del módulo debe coincidir con el de la constate (aunque se le podría poner otro nombre a la constante).

Si ahora se escribe "mates." en el intérprete de comandos y se pulsa el tabulador un par de veces:

```
Welcome to Node.js v12.18.3.
Type ".help" for more information.
> const mates = require('./mates');
undefined
> mates.
mates.__defineGetter__     mates.__defineSetter__     mates.__lookupGetter__     mates.__lookupSetter__
mates.__proto__
mates.constructor          mates.hasOwnProperty       mates.isPrototypeOf        mates.propertyIsEnumerable
mates.toLocaleString
mates.toString             mates.valueOf

mates.PI                   mates.resta                mates.suma
```

Se puede ver que se han importado los métodos del módulo mates. Se pueden invocar y trabajar con ellos:

```
> console.log(mates.PI + mates.suma(2,3));
8.14159
```

De la misma forma que se importan módulos desde la línea de comandos, se pueden importar desde un archivo. Por ejemplo, sea el archivo "calculos.js":

```
// Archivo calculos.js

// Se importa el módulo
const mates = require('./mates');

// Se hacen algunas operaciones con él:
var a = mates.suma(3, mates.PI);

console.log(a);
```

Se podrá ejecutar desde la línea de comandos con:

```
$ node calculos.js
```

Se recomienda consultar la API de "Node.js" para hacerse una idea de los módulos que vienen incluidos por defecto:

https://nodejs.org/docs/latest-v12.x/api/all.html

10.5 Ejecución síncrona y asíncrona

"Node.js" no facilita el trabajo con hilos y procesos. Javascript en el navegador no tiene hilos. Si estuviésemos escribiendo un servidor de páginas web, correríamos el peligro de que una página se quedase bloqueada hasta que no se haya servidor la anterior. Para evitar este problema y poder hacer ejecución de múltiples tareas, se usará la ejecución asíncrona. Básicamente se solicitará a "Node.js" hacer una tarea y se le pasará una función. "Node.js" ejecutará la función cuando el recurso esté disponible.

Por lo tanto se tendrán dos formas de realizar la ejecución:

- Síncrona: Es la ejecución normal de un script. Cada línea de código se ejecuta una tras otra.
- Asíncrona: Se solicita la ejecución de una función y "Node.js" la ejecutará cuando el recurso esté disponible o se genere algún evento.

Para verlo se trabajará con el módulo fs de "Node.js". El módulo fs permite leer o escribir archivos en el disco o leer los contenidos de un directorio (similar al comando ls de Linux). Para más información sobre el módulo fs, se puede consultar la documentación de "Node.js":

https://nodejs.org/download/docs/v12.18.3/api/fs.html

Viendo la documentación del módulo se dispone del método "fs.readFileSync(path[, options])" este método lee un archivo y nos devuelve el contenido en forma de texto (si se le indica una codificación, "uft-8"), la ejecución del programa se interrumpe hasta que no se haya leído el archivo. Se le debe pasar como parámetro la ruta, path, del archivo y la codificación. Por ejemplo si se ejecuta lo siguiente en el intérprete de comandos:

```
const fs = require('fs');
console.log('Se lee el archivo mates.js');
var texto = fs.readFileSync('mates.js', 'utf-8');
// La variable "texto" es un String con los contenidos del archivo
console.log(texto);
console.log('Fin de la lectura del archivo');
```

Mostrarán por pantalla los contenidos del archivo "mates.js".

En todo momento la ejecución del archivo ha sido síncrona. Por lo que primero se muestra el texto "Se lee el archivo...", después el texto del archivo, y por último "Fin de la lectura del archivo".

El módulo fs de "Node.js" también dispone del método "fs.readFile(path[, options], callback)" que hace una lectura asíncrona del archivo. Se le debe pasar la ruta del archivo, path, la codificación y una función, callback, que se ejecutará cuando los contenidos del archivo estén disponibles. La función callback tendrá dos argumentos err (indica si ha habido algún error en la lectura) y data (contenidos del archivo).

```
// Archivo lecturaasincrona.js

// Se importa el módulo fs
const fs = require('fs');

console.log('Se lee el archivo mates.js');

fs.readFile('mates.js', 'utf-8' ,(err, data) => {
   // Si hay un error en la lectura se lanza una excepción
   if (err) throw err;
   // Si el archivo se ha leído correctamente, se muestra el contenido
   console.log(data);
});

console.log('¿Fin de la lectura del archivo?');
```

Si la ejecución del archivo hubiese sido síncrona, se mostraría primero el mensaje "Se lee el archivo...", después el contenido del archivo y finalmente "Fin de la lectura del archivo".

Al ser asíncrona, primero se muestra el mensaje "Se lee el archivo...", después se hace la solicitud de lectura asíncrona del archivo, por lo que se le pasa la función que se ejecutará cuando el archivo esté disponible. Ahora, en función del estado del disco, del sistema operativo,... puede que se muestre primero el mensaje "¿Fin de la lectura del archivo?" y después el texto del archivo o viceversa.

La ventaja de la lectura asíncrona es que la ejecución del programa no se bloquea porque se tenga que leer el archivo, el script sigue ejecutándose. Esto es útil a la hora de escribir servidores, pues se podrán lanzar varias tareas que se ejecutarán según los recursos vayan estando disponibles.

De forma análoga, el módulo fs dispone de escritura de archivos síncrona y asíncrona:

```
// Archivo escritura.js

// Se importa el módulo fs
const fs = require('fs');

var texto="Este es el texto que se va a guardar en el archivo.";

// Se escribe el archivo de forma asíncrona con fs.writeFile(file, data[, options], callback)
// file es la ruta del archivo
// data es lo que se va a guardar en el archivo
// options generalmente se indica la codificación
// callback es la función que se ejecutará cuando el archivo haya sido guardado

fs.writeFile('salida1.txt', texto, 'utf-8' ,(err) => {
```

```
   // Si hay un error en la lectura se lanza una excepción
   if (err) throw err;
   console.log("El archivo salida1.txt se ha guardado correctamente");
});


// Se escribe el archivo de forma síncrona con fs.writeFileSync(file, data[, options])
// file es la ruta del archivo
// data es lo que se va a guardar en el archivo
// options generalmente se indica la codificación

try {
   fs.writeFileSync('salida2.txt', texto, 'utf-8');
   console.log("El archivo salida2.txt se ha guardado correctamente");
} catch(err) {
   console.log("Error al escribir el archivo salida2.txt");
}
```

10.6 Promise

Algunas funciones de ejecución asíncrona devuelven un objeto Promise. Una vez que se tiene el objeto Promise, se indica la función que se debe de ejecutar de forma asíncrona usando el método then:

```
let p = dirA.read(); // Se devuelve un objeto Promise
p.then( (e) => {console.log(e.name)} ); // Se indica la función a ejecutar de forma asíncrona con then
```

Aunque lo habitual es encadenar las peticiones:

```
dirA.read().then( (e) => {console.log(e.name)} );
```

Para ver un ejemplo real de Promise, se va a ver la forma que tiene el módulo fs de leer los contenidos de los directorios. El comando ls de Linux permite ver los contenidos de una carpeta. El módulo fs posee un mecanismo para leer los contenidos de un directorio mediante "fs.opendirSync(path[, options])" donde path es la ruta de la carpeta. options es un parámetro opcional que habitualmente se usará para indicar la codificación. Esta función devuelve un objeto del tipo fs.Dir que representa el directorio.

Se estudia primero la forma síncrona:

Para trabajar con opendirSync, primero se abre el directorio con opendirSync y después se van leyendo una a una las entradas con el método "fs.Dir.readSync()". El método "fs.Dir.readSync()" devuelve null si no hay mas contenidos en la carpeta (se ha leído todo) o un objeto "fs.Dirent" que representa al archivo. El objeto "fs.Dirent" indica el nombre del archivo, si es archivo o carpeta. Por ejemplo:

```
// Archivo lectura_dir.js

// Se importa el módulo fs
const fs = require('fs');

console.log('\nLectura síncrona:');

// Se abre la carpeta actual
var dirS = fs.opendirSync('.', 'utf-8');
//Se lee la primera entrada de la carpeta
```

```
var entrada = dirS.readSync();
while(entrada != null) { // Se ejecuta mientras no se llegue al final
   // Se comprueba si la entrada es un directorio
   if(entrada.isDirectory())
      console.log('Directorio: ' + entrada.name); // Con la propiedad name se lee el nombre del archivo o directorio
   // Se comprueba si es un archivo
   else if(entrada.isFile())
      console.log('Archivo: ' + entrada.name);
   entrada = dirS.readSync();
}
// Se cierra el directorio
dirS.close();

console.log('Fin del script');
```

También se puede trabajar de forma asíncrona. En este caso, en lugar de usar fs.Dir.readSync(), se usará fs.Dir.read() que devuelve un objeto del tipo Promise:

```
// Archivo lectura_dir_asincrona.js

// Se importa el módulo fs
const fs = require('fs');

console.log('\nLectura asíncrona:');

// Se abre la carpeta actual
var dirA = fs.opendirSync('.', 'utf-8');

// Esta es la función que se va a ejecutar en cada entrada
function leerEntradas(entrada) {
   if(entrada != null) {
      if(entrada.isDirectory())
         console.log('Directorio: ' + entrada.name);
      else if(entrada.isFile())
         console.log('Archivo: ' + entrada.name);
      // Se solicita leer la siguiente entrada
      dirA.read().then(leerEntradas);
   }
}

// Se ejecuta el método read que devuelve un objeto Promise
// y se le indica la función a ejecutar de forma asíncrona
dirA.read().then(leerEntradas);

// El resto del script continúa su ejecución
console.log('Fin del script');
```

Si se compara la salida de la ejecución síncrona con la asíncrona, se puede ver que el mensaje "Fin del script" se coloca en sitios diferentes.

10.7 async y await

async colocado delante de una función indica que la función va a resolver un objeto Promise:

```
// Archivo ej_async.js

async function f() {
```

```
   return 1;
}

f().then( (valor) => {console.log(valor)} );
```

Hay situaciones en las que se necesita convertir un Promise de asíncrono a síncrono. Para ello se usa la palabra clave await delante del objeto Promise. IMPORTANTE: Sólo funciona dentro de una función async.

En el siguiente ejemplo se transforma el objeto Promise, devuelto por fs.Dir.read(), de asíncrono a síncrono:

```
// Archivo lectura_dir_await.js

// Se importa el módulo fs
const fs = require('fs');

async function ls(ruta) {
   // Se abre la carpeta actual
   var dirS = fs.opendirSync(ruta, 'utf-8');
   //Se lee la primera entrada de la carpeta
   var entrada = await dirS.read();
   while(entrada != null) { // Se ejecuta mientras no se llegue al final
      // Se comprueba si la entrada es un directorio
      if(entrada.isDirectory())
         console.log('Directorio: ' + entrada.name); // Con la propiedad name se lee el nombre del archivo o directorio
      // Se comprueba si es un archivo
      else if(entrada.isFile())
         console.log('Archivo: ' + entrada.name);
      entrada = await dirS.read();
   }
   // Se cierra el directorio
   dirS.close();
}

ls('.');
```

11 Módulos y NPM

Se recomienda consultar la API de "Node.js" para hacerse una idea de los módulos que vienen incluidos por defecto:

https://nodejs.org/docs/latest-v12.x/api/all.html

Esta API por defecto está muy limitada, pero aún así permite crear un servidor HTTP de forma sencilla.

"Node.js" tiene una herramienta que permite descargar más módulos llamada **npm**. La cantidad de módulos disponibles es inmensa. Estos se pueden consultar en:

https://www.npmjs.com/

Los módulos que se pueden instalar se pueden instalar de dos formas, en la carpeta actual y globalmente. Si se instalan globalmente, se podrá acceder a dicho módulo en cualquier carpeta de

nuestro disco duro. Si se instala en la carpeta actual, sólo se podrá acceder a dicho módulo cuando estemos en dicha carpeta.

Para instalar un módulo localmente se usará el comando:

```
npm install "nombre del módulo"
```

Por ejemplo, para instalar el módulo "electron" en la carpeta actual:

```
npm install electron
```

Se creará una carpeta llamada "node_modules". Dentro de esta carpeta, se encontrará el módulo descargado y sus dependencias. Si se consulta la carpeta del módulo, a veces se localiza el código fuente y su documentación.

Para instalarlo globalmente se usará la opción "-g":

```
npm install electron -g
```

Cuando se instala globalmente, puede que también instalen ejecutables que se podrán usar. Por ejemplo, si se instala electron globalmente se tendrá disponible el comando "electron". Pruebe a instalar electron globalmente y a ejecutar el comando electron.

Los módulos instalados estarán disponibles con require:

```
var electron = require('electron');
```

Es recomendable, antes de hacer cualquier tarea, ver los módulos disponibles en https://www.npmjs.com/. Por ejemplo, si se desea ver los archivos de una carpeta ya se ha visto fs.opendirSync, pero el módulo "glob" tiene mucha más potencia y sencillez. Entre los módulos recomendados se tienen:

- express: Es una herramienta para crear sitios web.
- mysql: Conexión con bases de datos MySQL.
- mongodb: Conexión con bases de datos MongoDB.
- Jade: Plantillas para crear páginas web.
- browserify: Javascript del navegador no dispone de require, esta utilidad crea un solo archivo Javascript juntando todos los archivos a través de sus require.
- electron: Es un navegador web que permite crear aplicaciones de escritorio. Microsoft Visual Studio o Atom son ejemplos de aplicaciones creadas con electron.

11.1 Creando proyectos "Node.js"

Para crear un proyecto con "Node.js" se creará una carpeta en el sitio donde se vaya a almacenar el proyecto. Después se creará el archivo "package.json", para ello se usará la herramienta "npm init". Sólo hay que ejecutar en un terminal:

```
npm init
```

Un asistente nos irá preguntando características del proyecto que se está creando. Por ejemplo:

```
$ mkdir ejemplo
$ cd ejemplo/
/ejemplo$ npm init
```

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install pkg` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (ejemplo) Ejemplo
Sorry, name can no longer contain capital letters.
package name: (ejemplo) ejemplo
version: (1.0.0) 0.0.1
description: Ejemplo de proyecto nodejs
entry point: (index.js) main.js
test command:
git repository:
keywords: ejemplo, node
author: Yo
license: (ISC) BSD-3-Clause
About to write to /media/lucas/Lucas/Publico/Asignaturas/Programacion-multimedia/Melonjs-2020-08-
13/ejemplo/package.json:

{
  "name": "ejemplo",
  "version": "0.0.1",
  "description": "Ejemplo de proyecto nodejs",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ejemplo",
    "node"
  ],
  "author": "Yo",
  "license": "BSD-3-Clause"
}


Is this OK? (yes) yes
```

El asistente muestra el archivo "package.json" que se creará. Los campos de dicho archivo son sencillos de entender, nombre, versión, palabras clave,... Pero hay otros campos, como scripts, que requieren un explicación más detallada.

11.2 Campo scripts de package.json

El campo scripts sirve para ejecutar comandos en línea de comandos. Por ejemplo, si se modifica el archivo de la siguiente forma:

```
{
  "name": "ejemplo",
  "version": "0.0.1",
  "description": "Ejemplo de proyecto nodejs",
  "main": "main.js",
  "scripts": {
```

```
    "test": "echo \"Error: no test specified\" && exit 1",
    "listado": "ls -l"
  },
  "keywords": [
    "ejemplo",
    "node"
  ],
  "author": "Yo",
  "license": "BSD-3-Clause"
}
```

Se añade un script, al que se ha llamado "listado" y que ejecutará el comando por línea de comandos "ls -l".

Para ejecutar los scripts se ejecutará por línea de comandos:

```
npm run "nombre del script"
```

Por ejemplo, para ejecutar el script "listado" que se acaba de añadir:

```
npm run listado
```

Pero será más habitual a crear, por ejemplo, un script llamado "build" que ayuda a construir la aplicación y otro llamado "start" para lanzar la aplicación. Veámoslo con un ejemplo:

Supongamos que se crean los archivos "build.js" y "server.js" en nuestro proyecto. Supongamos que "build.js" crea unas páginas web a partir de unas plantillas y que "server.js" es el servidor HTTP que se ha visto en ejemplos anteriores. Se podría modificar "package.json" para añadir lo siguiente:

```
{
  "name": "ejemplo",
  "version": "0.0.1",
  "description": "Ejemplo de proyecto nodejs",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "node build.js",
    "start": "npm run build && node server.js"
  },
  "keywords": [
    "ejemplo",
    "node"
  ],
  "author": "Yo",
  "license": "BSD-3-Clause"
}
```

Si se ejecuta el comando:

```
$ npm run build
```

Se ejecutará por línea de comandos:

```
> node build.js
```

Si se ejecuta el comando:

```
$ npm run start
```

Se ejecutará por línea de comandos primero "npm run build" que a su vez ejecuta:

```
> node build.js
```

Se ejecutará también "node server.js" que lanzará nuestro servidor web para ver las páginas.

11.3 Campo dependencies de package.json

Cuando se crea un proyecto, puede que se necesiten instalar módulos, usando npm install nombre, para instalar las dependencias del proyecto. Por ejemplo, si nuestro proyecto va a usar el módulo electron, sería cómodo indicarlo en "package.json" y que con un comando se instalasen de forma automática. Para indicar las dependencias se usará el campo dependencies. Por ejemplo, para indicar que nuestro proyecto depende de electron y del paquete tex2max:

```
{
 "name": "ejemplo",
 "version": "0.0.1",
 "description": "Ejemplo de proyecto nodejs",
 "main": "main.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "node build.js",
  "start": "electron main.js"
 },
 "keywords": [
  "ejemplo",
  "node"
 ],
 "author": "Yo",
 "license": "BSD-3-Clause",
 "dependencies": {
  "electron": "^2.0.4",
  "tex2max": "^1.3.0"
 }
}
```

Como se puede ver en dependencies se debe indicar el nombre del paquete y la versión que se necesita para que el proyecto funcione.

Para instalar las dependencias se usará:

```
npm install
```

Al ejecutarlo instalará los paquetes solicitados.

En el caso de querer que se ejecute algún comando al finalizar la instalación de los paquetes se puede añadir el campo "postinstall" a los scripts:

```
{
 "name": "ejemplo",
 "version": "0.0.1",
 "description": "Ejemplo de proyecto nodejs",
 "main": "main.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "node build.js",
  "start": "electron main.js",
  "postinstall": "node install.js"
```

```
 },
 "keywords": [
   "ejemplo",
   "node"
 ],
 "author": "Yo",
 "license": "BSD-3-Clause",
 "dependencies": {
   "electron": "^2.0.4",
   "tex2max": "^1.3.0"
 }
}
```

Cuando se ejecute "npm install", al finalizar la instalación de las dependencias se ejecutará el comando asociado a "postinstall" que en este caso es "node install.js". Si se escribe un archivo "install.js", se puede usar para tareas como instalar ejecutables o procesar plantillas.

Se recomienda al lector hacer un ejercicio creando un proyecto con algunas dependencias y scripts.