Índice

2. Sintaxis, funciones y componentes

Antes de comenzar con *JavaScript* es necesario fijar una serie de conocimientos previos. A lo largo del manual se van a utilizar páginas **HTML**, por lo que se debe saber las etiquetas básicas, la estructura de una página web, uso de listas y capas. Junto a **HTML** se definirá su estilo mediante **CSS**, por lo que es necesario conocer su uso básico y cómo se aplica un estilo a una etiqueta, una clase o un identificador.

Dentro del mundo de la programación *JavaScript* tiene mala fama. Gran parte se debe a que se trata de un **lenguaje débilmente tipado**, lo que permite usar variables sin declarar. Además, al tratarse de un **lenguaje interpretado**, no hay compilador que te diga que hay código erróneo hasta la su ejecución, aunque esto ya no representa un problema gracias a los editores de código. Realmente, *JavaScript* ofrece mucha flexibilidad y las malas críticas vienen más por el desconocimiento del lenguaje que por defectos del mismo.

Contrario a lo que el nombre sugiere, *JavaScript* tiene poco que ver con *Java*. La similitud fue una decisión de marketing allá por el año 1995 cuando *Netscape* introdujo el lenguaje en el navegador.

Una de las versiones más extendidas de *JavaScript* moderno es la llamada por por muchos ***Javascript ES6*** (*ECMAScript* 6), también llamado ***ECMAScript* 2015** o incluso por algunos llamado directamente *Javascript* 6.

Dada su popularidad, el lenguaje ha sido portado a otros ámbitos entre los que destaca el popular ***NodeJS*** que permite la ejecución de *JavaScript* como como lenguaje escritorio y lenguaje servidor.

2.1. *Strict mode*

El modo estricto elimina características del lenguaje, lo que simplifica los programas y reduce la cantidad de errores que pueden contener. Por ejemplo, usar variables que no se han declarado previamente, declarar funciones donde algún parámetro está repetido, un objeto donde dos propiedades tengan el mismo nombre, etcétera. De este modo se pueden evitar ciertas características que han sido muy criticadas para este lenguaje.

Para activar el modo estricto hay que introducir la cadena `use strict` como primera línea del ámbito que necesite esta característica. Si ese ámbito es el global habrá que incluirlo en la primera línea del fichero que contiene el código.

Por ejemplo, si sólo es necesario activar el modo dentro de una función se escribirá:

```
function modoEstricto() {
    "use strict";
    // resto de la función
  }
```

Esto significa que el código interno de la función se ejecutará con el subconjunto estricto del lenguaje, mientras que otras funciones puede que hagan uso del conjunto completo.

El objetivo es que en el futuro sólo se soportará el modo estricto, con lo que ES5 fue una versión transicional en la que se anima (pero no obliga) a escribir código en modo estricto.

2.2. Uso en el navegador

Para utilizar *JavaScript* dentro de una página web usaremos la etiqueta **<script>** . Mediante esta etiqueta se puede incluir el código:

- en el propio documento **HTML**:

```
<script>
   //Instrucciones JavaScript
</script>
```

- en un archivo externo (con extensión **.js**), de modo que se pueda reutilizar entre varios documentos:

```
<script src="ficheroJavaScript.js"></script>
```

- como parte de un manejador de eventos, como puede ser **onClick** o **onMouseOver**

```
<button onclick="nombreFuncionJavaScript()" />
```

- como parte de una URL mediante el pseudo-protocolo **javascript**:

```
<a href="javascript:nombreFuncionJavaScript()">Validar</a>
```

Normalmente las referencias *JavaScript* se incluyen mediante la etiqueta **<script>** en la cabecera del documento **HTML**, pero si es necesario que el código se ejecute al cargar la página es conveniente ponerlo antes de cerrar la etiqueta **<body>** para asegurarnos que se ha cargado todo el **DOM**.

2.3. Datos y variables

Durante el desarrollo de este manual se utilizará el estilo ***lowerCamelCase*** para identificar a las variables. Más información aquí .

Para declarar las variables en *JavaScript* se puede utilizar **let** o **var**, según el ámbito donde deba ser accesible:

- **let** permite declarar una variable que sea accesible únicamente dentro del bloque donde se ha declarado (llamamos bloque al espacio delimitado por **{ }**).
- **var** permite declarar una variable que sea accesible por todos los lugares de la función donde ha sido declarada. Si una variable con **var** se declara fuera de cualquier función el ámbito de esta son todas las funciones del código.
- sin declarar, *JavaScript* permite usar variables no declaradas. Si ocurre esto, será equivalente a declararlas con **var** fuera del código, es decir, serán variables accesibles por cualquier función. Esto no es posible si se está utilizando **strict mode**.

Y como ejemplo el siguiente código:

```
function ejemplo(){
    ejemploFeo=3; // Equivale a declararla fuera de la función como var.
    if (ejemploFeo === 3){
        var variable1 = 1;
        let variable2 = 2;
    };
```

```
    console.log(variable1); // variable1 existe en este lugar.
    console.log(variable2); // variable2 no existe en este lugar.
};
```

2.4. *Hoisting*

Una cosa muy interesante de los lenguajes débilmente tipados e interpretados es el concepto de *hoisting*, el cual eleva las declaraciones encontradas en un bloque a la primera línea. Esta característica no sólo funciona con variables, sino también con funciones declaración (como se verá en lo sucesivo).

Si se escribe el siguiente código:

```
"use strict";
var a = "global";
console.log(b); // undefined
var b = 5;
```

La variable `b` no se había declarado y pese a ello, aún usando el modo estricto, el intérprete no lanza ninguna excepción porque la declaración se eleva al principio del bloque, no así la asignación, que permanece en su lugar y por eso no toma el valor `5`, sino `undefined`.

2.5. Tipos de datos

Los principales tipos de datos primitivos en *JavaScript* son:

- Numéricos (***number***) puede contener cualquier tipo de número real o entero.
- Booleanos (***boolean***) puede contener uno de los siguientes valores: `true`, `false`, 1 y 0.
- Indefinidos (***undefined***) valor por defecto si no se le asigna uno.
- Cadenas (***string***) cualquier combinación de caracteres **UTF-16** (letras, números, signos especiales y espacios). Las cadenas se delimitan mediante comillas dobles o simples. Se puede incluir comillas dobles dentro de una cadena creada con comillas simples, así como comillas simples dentro de una cadena creada con comillas dobles. Para concatenar cadenas puede usarse el operador `+`, pero existe una mejor manera de hacerlo a través de los ***template literals*** (más información aquí). Es recomendable aprender su uso ya que es el método que utiliza `React` para las vistas de sus componentes.

Todos los valores que no son de un tipo **primitivo** (básico) son considerados **objetos**: arrays, funciones, valores compuestos, etcétera. Esta distinción es muy importante porque los valores primitivos y los valores objetos se comportan de distinta forma cuando son asignados y cuando son pasados como parámetro a una función. La principal diferencia es que los tipos primitivos se pasan como parámetros, se copian y se comparan por valor; mientras que los objetos lo hacen por referencia. Más información aquí.

El operador `typeof` devuelve una cadena que identifica el tipo del operando. Así pues,

```
typeof 94.8; // Que devuelve 'number'
typeof "Feo"; // Que devuelve 'string'
let edad=23, nuevaEdad, incremento=4;
const nombre="Feo Muy";
console.log(typeof incremento === "number");
nuevaEdad=edad+incremento;
```

```
console.log(nombre+ " tras "+incremento +" años tendrá "+ nueva_edad);
```

Los posibles valores de **typeof** son **number**, **string**, **boolean**, **undefined**, **function** y **object**. El problema viene cuando se hace esto:

```
typeof null // Que indica 'object'.
```

y en vez de recibir **null** dice que es un **object**. Es decir, si se le pasa un array o *null* el resultado es idéntico; **object**, lo cual es incorrecto. Para comprobar si es **null** es mejor usar el operador identidad:

```
valorNulo === null // true
```

2.6. Coerción

JavaScript es un lenguaje de tipado blando, es decir, al declarar una variable no se le asigna un tipo aunque internamente sí maneje tipos de datos.

En determinados momentos resulta necesario convertir un valor de un tipo a otro. Esto en *JavaScript* se llama coerción y puede ocurrir de forma **implícita** o podemos forzarlo de forma **explícita**.

```
let numero = 5;
console.log(numero);
```

En este código ocurre una coerción implícita de número (que es de tipo **number**) a un tipo **string**, de modo que puede ser impreso por consola. Se podría realizar la conversión de forma explícita de la siguiente forma:

```
console.log(numero.toString());
```

Las coerciones implícitas ocurren muy a menudo en *JavaScript* aunque muchas veces no se sea conscientes de ello. Resulta muy importante entender cómo funcionan para poder deducir cuál será el resultado de una comparación.

```
let a = "2", b = 5;
console.log( typeof a + " " + typeof b); // string number
console.log( a + b ); // muestra 25
```

En los lenguajes de tipado duro (como *Java*) se prohíbe realizar operaciones entre distintos tipos de datos. Sin embargo, *JavaScript* lo permite siguiendo una serie de reglas:

- tiene el operador **===** y **!==** para realizar comparaciones estrictas (identidad), pero no posee esos operadores para desigualdades (**>**, **<**, **>=**, **<=**)
- si es posible, *JavaScript* prefiere hacer coerciones a tipo **number** por encima de otros tipos básicos. Por ejemplo, la expresión **("15" < 100)** se resolverá como **true** porque cambiará **"15"** de tipo **string** por **15** de tipo **number**. Si se convierte **"15"** a **string**, al compararlo con **"100"** la expresión se resolvería como **false**
- a la hora de hacer coerción a **boolean** los siguientes valores se convertirán en **false**: **undefined**, **null**, **0**, "", **NaN**. El resto de valores se convertirán en **true.**

Ejemplo donde no se hace coerción :

```
var x = "10";
var y = "9";
```

```
console.log(x < y);  // true, los dos son String y los compara como cadena.
```

Ejemplo de coerción con **undefined**

```
let altura; // Variable no definida.
console.log(altura ? true : false); // Al no estar definido, false.
```

Al realizar comparaciones, si se usa **==** o **!=** para comparar los datos, *JavaScript* realiza coerción. Si se quiere que la comparación no convierta tipos y sólo sea cierta si son del mismo tipo, se debe usar **===** o **!==**. Esta es una buena práctica muy recomendada para que estas conversiones no jueguen malas pasadas.

2.7. Constantes

Las constantes son elementos que permiten almacenar un valor el cual permanece invariable (permanece constante). Para declarar constantes se utiliza la instrucción **const**. Su ámbito es el mismo que el de **let**, es decir, sólo son accesibles en el bloque que se han declarado.

```
const PI=3.1416;
console.log(PI);
PI=3; // Esto falla.
```

Aunque es posible definir *arrays* y objetos usando **const**, no es recomendable hacerlo ya que es posible que su uso no sea el que se espera. Un *array* o un objeto declarado como constante mantendrá invariable su estructura no así su contenido.

Por ejemplo, al declarar un *array* realmente lo que ocurre es que la variable almacena su dirección de memoria. Si se declara usando **const**, lo que ocurre es que no puede cambiarse esa dirección de memoria, pero permitirá cambiar sus valores.

```
const miArray=[1,2,3]
console.log(miArray[0]); // Muestra el valor 1.
miArray[0]=4;
console.log(miArray[0]); // Muestra el valor 4.
miArray=[]; // Esto falla.
```

2.8. Operadores

Combinando variables y valores se forman expresiones complejas que son una parte clave en la creación de programas. Para formular expresiones se utilizan los operadores.

2.8.1. Operadores de asignación

Los operadores de asignación se utilizan para asignar valores a las variables. Algunos de ellos, además de asignar el valor también incluyen operaciones:

Operador	Descripción
=	Asigna a la variable de la parte izquierda el valor de la parte derecha.
+=	Suma los operandos izquierdo y derecho y asigna el resultado al operando izquierdo.
-=	Resta el operando derecho del izquierdo y asigna el resultado al operando izquierdo.
*=	Multiplica ambos operandos y asigna el resultado al operando izquierdo.
/=	Divide ambos operandos y asigna el resultado al operando izquierdo.

```
let num1=3;
let num2=5;
num2+=num1;
num2-=num1;
num2*=num1;
num2/=num1;
num2%=num1;
```

2.8.2. Operadores aritméticos

Se utilizan para realizar cálculos aritméticos.

Operador	Descripción
+	Suma.
-	Resta.
*	Multiplicación.
/	División.
%	Calcula el resto de una división entera.

Además de estos operadores, también existen operadores aritméticos unitarios: incremento (**++**), disminución (**--**) y la negación unitaria (**-**).

Los operadores de incremento y disminución pueden estar tanto delante como detrás de una variable con distintos matices en su ejecución. Estos operadores aumentan o disminuyen en uno el valor de una variable.

Operador	Descripción (Suponiendo x=5)
y = ++x	Primero el incremento y después la asignación **x=6**, **y=6.**
y = x++	Primero la asignación y después el incremento **x=6**, **y=5.**
y = --x	Primero el decremento y después la asignación **x=4**, **y=4.**
y = x--	Primero la asignación y después el decremento **x=4**, **y=5.**

y =-x	Se asigna a **y** el valor negativo de **x**, pero el valor de la variable **x** no varía **x=5**, **y= -5.**

```
let num1=5, num2=8, resultado1, resultado2;
resultado1=((num1+num2)*200)/100;
resultado2=resultado1%3;
resultado1=++num1;
resultado2=num2++;
resultado1=--num1;
resultado2=num2--;
resultado1=-resultado2;
```

2.8.3. Operadores de comparación

Utilizados para comparar dos valores entre sí. Como valor de retorno se obtiene siempre un valor booleano: **`true`** o **`false`**.

Operador	Descripción
`===`	Compara dos elementos, incluyendo su tipo de dato. Si son de distinto tipo, no realiza conversión y devuelve **`false`** ya que siempre los considera diferentes. **Uso recomendado.**
`!==`	Compara dos elementos, incluyendo su tipo interno. Si son de distinto tipo, no realiza conversión y devuelve **`true`** ya que siempre los considera diferentes. **Uso recomendado.**
`==`	Devuelve el valor **`true`** cuando los dos operandos son iguales. Si los elementos son de distintos tipos, realiza una conversión. **No está recomendado su uso.**
`!=`	Devuelve el valor **`true`** cuando los dos operandos son distintos. Si los elementos son de distintos tipos, realiza una conversión. **No está recomendado su uso.**
`>`	Devuelve el valor **`true`** cuando el operando de la izquierda es mayor que el de la derecha.
`<`	Devuelve el valor **`true`** cuando el operando de la derecha es menor que el de la izquierda.
`>=`	Devuelve el valor **`true`** cuando el operando de la izquierda es mayor o igual que el de la derecha.
`<=`	Devuelve el valor **`true`** cuando el operando de la derecha es menor o igual que el de la izquierda.

```
let a=4;b=5,c="5";
console.log("El resultado de la expresión 'a==c' es igual a "+(a==c));
console.log("El resultado de la expresión 'a===c' es igual a "+(a===c));
console.log("El resultado de la expresión 'a!=c' es igual a "+(a!=c));
console.log("El resultado de la expresión 'a!==c' es igual a "+(a!==c));
console.log("El resultado de la expresión 'a==b' es igual a "+(a==b));
console.log("El resultado de la expresión 'a!=b' es igual a "+(a!=b));
console.log("El resultado de la expresión 'a>b' es igual a "+(a>b));
console.log("El resultado de la expresión 'a<b' es igual a "+(a<b));
console.log("El resultado de la expresión 'a>=b' es igual a "+(a>=b));
console.log("El resultado de la expresión 'a<=b' es igual a "+(a<=b));
```

2.8.4. Operadores lógicos

Los operadores lógicos se utilizan para el procesamiento de los valores booleanos. A su vez el valor que devuelven también es booleano: **`true`** o **`false`**.

Operador	Descripción
`&&`	Y lógica. El valor de devolución es **`true`** cuando ambos operandos son verdaderos.
`||`	O lógica. El valor de devolución es **`true`** cuando alguno de los operandos es verdadero o lo son los dos.
`!`	No lógico. Si el valor es **`true`**, devuelve **`false`** y si el valor es **`false`**, devuelve **`true.`**

Se muestra el resultado de distintas operaciones realizadas con operadores lógicos.

```
console.log("El resultado de la expresión 'false&&false' es igual a "+(false&&false));
console.log("El resultado de la expresión 'false&&true' es igual a "+(false&&true));
console.log("El resultado de la expresión 'true&&false' es igual a "+(true&&false));
console.log("El resultado de la expresión 'true&&true' es igual a "+(true&&true));
console.log("El resultado de la expresión 'false||false' es igual a "+(false||false));
console.log("El resultado de la expresión 'false||true' es igual a "+(false||true));
console.log("El resultado de la expresión 'true||false' es igual a "+(true||false));
console.log("El resultado de la expresión 'true||true' es igual a "+(true||true));
console.log("El resultado de la expresión '!false' es igual a "+(!false));
```

Más información sobre comparadores y expresiones aquí .

2.9. Estructuras de control

2.9.1. Instrucciones `if/else`

Para controlar el flujo de información en los programas *JavaScript* existen una serie de estructuras condicionales y bucles que permiten alterar el orden secuencial de ejecución. Estas son las instrucciones **`if`** y **`else`**.

La instrucción **`if`** permite la ejecución de un bloque de instrucciones u otro en función de una condición.

```
if (condición) {
    // Bloque de instrucciones que se ejecutan si la condición se cumple.
}
else{
    // Bloque de instrucciones que se ejecutan si la condición no se cumple.
}
```

Las llaves pueden omitirse si el bloque de instrucciones contiene una sola línea. De igual modo es posible omitir el punto y coma del final de esa instrucción. Ambas acciones **no son recomendables** ya que , en ocasiones, esto no puede realizarse y puede ocasionar algunos problemas.

Puede existir una instrucción **`if`** que no contenga la parte **`else`**.

```
let diaSem;
diaSem=prompt("Introduce el día de la semana ", "");
if (diaSem === "domingo")
{
    console.log("Hoy es festivo");
}
else // Al no tener {}, es un "bloque de una instrucción".
    console.log("Hoy no es domingo, a descansar!!");

let edadAna,edadLuis;
// Se usa parseInt para convertir a entero.
edadAna=parseInt(prompt("Introduce la edad de Ana",""));
edadLuis=parseInt(prompt("Introduce la edad de Luis",""));
if (edadAna > edadLuis){
    console.log("Ana es mayor que Luis.");
    console.log("Ana tiene "+edadAna+" años y Luis "+ edadLuis);
}
else{
    console.log("Ana es menor o de igual edad que Luis.");
    console.log("Ana tiene "+edadAna+" años y Luis "+ edadLuis);
}
```

2.9.2. Estructura repetitivas (bucles)

Bucle `for`

Cuando la ejecución de un programa llega a un bucle **`for`**:

- lo primero que hace es ejecutar la **inicialización del índice**, que sólo se ejecuta una vez,
- a continuación analiza la **condición de prueba** y si esta se cumple ejecuta las instrucciones del bucle,
- cuando finaliza la ejecución de las instrucciones del bucle se realiza la **modificación del índice**, se retorna a la cabecera del bucle **`for`** y se realiza de nuevo la condición de prueba,
- si la condición se cumple se ejecutan las instrucciones y si no se cumple la ejecución continúa en las líneas de código que siguen posteriores al bucle.

```
for (Inicialización del índice; Condición de prueba; Modificación en el índice){
    // Instrucciones...
}

for (i=2;i<=30;i+=2) {
    console.log(i);
}
console.log("Se han escrito los números pares del 2 al 30.");

let aux=1;
for (i=2;i<=3000;i*=2) {
    console.log("2 elevado a "+aux+" es igual a "+i);
    aux++;
}
console.log("Se han escrito las potencias de 2 menores de 3000.");
```

Bucle `while`

Con el bucle **`while`** se pueden ejecutar un grupo de instrucciones mientras se cumpla una condición:

- si la condición nunca se cumple, entonces tampoco se ejecuta ninguna instrucción,
- si la condición se cumple siempre, nos veremos inmersos en el problema de los **bucles infinitos**, que pueden llegar a colapsar el navegador o incluso el ordenador. Por esa razón es muy importante que la condición deba dejar de cumplirse en algún momento.

```
while (condición){
    // Instrucciones...
}

let i=2;
while (i<=30) {
    console.log(i);
    i+=2;
}
```

```
console.log("Ya se han mostrado los números pares del 2 al 30.");

let auxclave="";
while (auxclave!=="viva"){
    auxclave=prompt("Introduce la clave: ","claveSecreta")
}
console.log("Has acertado la clave.");
```

Bucle `do-while`

La diferencia del bucle **`do-while`** frente al bucle **`while`** reside en el momento en que se comprueba la condición: el bucle **`do-while`** no la comprueba hasta el final, es decir, después del cuerpo del bucle, lo que significa que este bucle se realizará una vez como mínimo aunque no se cumpla la condición.

```
do {
    // Instrucciones...
} while(condición);

let auxclave;
do {
    auxclave=prompt("Introduce la clave: ","vivaYo")
} while (auxclave!=="EstaEslaclave")
console.log("Has acertado la clave.");
```

Instrucciones `BREAK` y `CONTINUE`

En los bucles **`for`**, **`while`** y **`do-while`** se pueden utilizar las instrucciones **`break`** y **`continue`** para modificar el comportamiento del bucle. La instrucción **`break`** dentro de un bucle hace que éste se interrumpa inmediatamente aún cuando no se haya ejecutado todavía el bucle completo. Al llegar a esta instrucción sale del bucle y el programa se sigue desarrollando inmediatamente.

```
let auxclave=true;
let numveces=0;
//Mientras no introduzca la clave y no se pulse Cancelar.
while (auxclave !== "anonimo" && auxclave){
    auxclave=prompt("Introduce la clave ","");
    numveces++;
    if (numveces === 3)
        break;
}
if (auxclave=="SuperClave"{
    console.log("La clave es correcta.");
}else{
    console.log("La clave no es correcta correcta.");
}
```

El efecto que tiene la instrucción **`continue`** en un bucle es el de hacer retornar a la secuencia de ejecución a la cabecera del bucle volviendo a ejecutar la condición o a incrementar los índices cuando sea un bucle **`for`**. Esto permite saltarse iteraciones del bucle, pero no sale de él.

```
let i;
for (i=2;i<=50;i+=2){
    if ((i%3)===0)
        continue;
    console.log(i);
}
```

En este punto es necesario introducir un par de operadores que funcionan como sentencias de flujo. Su uso es muy común ya que son utilizadas en la programación reactiva. Además, acortan las líneas de código escritas y amplían su legibilidad.

- **operador ternario**, es una estructura alternativa simple en una sola línea que se puede utilizar casi cualquier parte. La sintaxis es la siguiente: **`condición ? expr1 : expr2`** . Si la condición se evalúa a **`true`** el operador retorna **`expr1`**, de lo contrario retornará **`expr2`**.

  ```
  let esMayor = 18;
  console.log(`El discente es ${esMayor>17  ?  "mayor de edad" : "menor de edad"}`);
  ```

  Más información Aquí .

- ***nullish coalescing operator***, sirve para comprobar si una elemento tiene valor **`null`** o **`undefined`**: **`valor ?? expr1`**. Si valor es **`null`** o **`undefined`** devolverá **`expr1`**, si no lo es entonces devolverá **`valor`**.

  ```
  let nombre;
  console.log(`Hola ${nombre ?? “Desconocido”}`);
  ```

2.10. Funciones

Las funciones en *JavaScript* son objetos, y como tales, se pueden usar como cualquier otro valor. Las funciones pueden almacenarse en variables, objetos y *arrays*. Se pueden pasar como argumentos a funciones y una función a su vez puede devolver una función (ella misma u otra). Además, como objetos que son, pueden tener métodos.

Pero lo que hace especial a una función respecto a otros tipos de objetos es que las funciones pueden invocarse. Se crean con la palabra clave **`function`** junto a los parámetros sin tipo de datos y dentro de una pareja de paréntesis. El nombre de la función es opcional.

JavaScript no produce ningún error de ejecución si el número de argumentos y el de parámetros no coincide. Si hay demasiados valores de argumentos, los argumentos de sobra se ignoran. Por contra, si hay menos argumentos que parámetros, los parámetros que han quedado sin asignar tendrán el valor **`undefined`**. No se realiza ninguna comprobación de tipos, con lo que se puede pasar cualquier valor como parámetro.

Además, dentro de una función podemos invocar a otra función que definimos en el código a posteriori con lo que no tenemos ninguna restricción de declarar las funciones antes de usarlas. Pese a no tener restricción es una buena práctica de código que las funciones que dependen de otras se coloquen tras ellas.

Por último, para devolver cualquier valor dentro de una función usaremos la instrucción **`return`** , la cual es opcional. Si una función no hace **`return`** , el valor devuelto será **`undefined`**.

Las funciones en *JavaScript* admiten **parámetros por defecto**. Es posible asignarles un valor durante la declaración y, si no se especifica ninguno en la llamada, el parámetro tendrá ese valor por defecto.

2.10.1. Función declaración

Si al crear una función le asignamos un nombre se conoce como una función declaración.

```
function miFuncion(param1, param2=0) {
    // Instrucciones.
    // Si a param2 no se le asigna valor en la llamada, valdrá 0.
    return variable; // Si no se especifica devuelve undefined.
}
```

Las variables declaradas dentro de la función **no serán visibles desde fuera de la función**. Además, los **parámetros se pasan por copia**, y aquí viene lo bueno, **se pueden pasar funciones como parámetro de una función** y una función puede devolver otra función.

```
function suma(alfa, beta) {
    return alfa + beta;
}

function calculando(gamma, delta, fn) {
    return fn(gamma, delta);
}

var epsilon = calculando(3, 4, suma);
```

2.10.2. Función anónima (expresión)

Otra característica de las funciones de *JavaScript* es que una función se considera un valor. De este modo, se puede declarar una función anónima y asignarla a una variable, lo que se conoce como función expresión.

```
var miFuncionExpresion = function (param1, param2) {
    // Instrucciones.
}; // Es una expresión luego tiene que terminar con punto y coma.
```

Así pues, el mismo código del ejemplo anterior quedaría así:

```
var suma = function (alfa, beta) {
    return alfa + beta;
};

var calculando = function (gamma, delta, fn) {
    return fn(gamma, delta);
};

var epsilon = calculando(3, 4, suma);
```

Las funciones expresión se pueden invocar inmediatamente lo que hace que sean muy útiles cuando se tiene un bloque que se va a utilizar una única vez.

```
(function() {
    // Instrucciones.
})(); // Invoca la función inmediatamente.
```

Cabe destacar una diferencia importante entre estos tipos de funciones, y es que las funciones declaración **se cargan antes de cualquier código**, con lo que el motor *JavaScript* permite ejecutar una llamada a esta función incluso si está antes de su declaración. En cambio, con las funciones anónimas, **se cargan a medida que se avanza el *script*** y no van a permitir realizar una llamada a la función hasta que sea declarada por lo que se deben colocar antes del resto de código que quiera invocar dicha función.

```
cantar();
estribillo(); // TypeError: undefined
function cantar() {
    console.log("¿Qué puedo hacer?");
}
var estribillo = function() {
    console.log("He pasado por tu casa 20 veces.");
};
```

2.10.3. Funciones flecha (*arrow functions*)

Una función flecha (*arrow function*) es una alternativa compacta al uso de funciones declaración y anónimas. En realidad es una reducción de las primeras. Este tipo de funciones tienen sus limitaciones y deben ser utilizadas sólo en algunos contextos donde sean útiles y no son adecuadas para ser utilizadas como métodos.

Soporta varias sintaxis y son utilizados con la función **`map`** y **`reduce`** de los nuevos *arrays* de *JavaScript* y también como *callbacks* o en comunicaciones asíncronas con AJAX, pero en este apartado se tratará para el paso de funciones como parámetro.

```
() => {sentencias} // Sin parámetros.
(parametro1, parametro2, ...) => {sentencias} // Con parámetros.
parámetro => sentencia // Versión simplificada (evitar a toda costa).
```

La función **`calculando`** del ejemplo del apartado anterior recibe una función como parámetro. *JavaScript* permite incluir el código de la función parámetro en la llamada:

```
var epsilon = calculando(3, 4, function suma(alfa, beta) {
    return alfa + beta;
});
```

También es posible utilizar funciones anónimas si la función parámetro sólo se utiliza una vez o si está diseñada para esta tarea en concreto:

```
var epsilon = calculando(3, 4, function (alfa, beta) {
    return alfa + beta;
});
```

Para simplificar todo esto, es posible utilizar las funciones flecha compactando el código:

```
var epsilon = calculando(3, 4, (alfa, beta) => { alfa + beta; });
```

Se elimina las palabras reservadas **`function`** y **`return`** y además, al ser una única línea, es posible quitar las llaves aunque al igual que con el punto y coma siempre es recomendable dejarlas.

El uso de las funciones flecha se utilizará de forma profusa a medida que se avance en el módulo. Es buena idea familiarizarse con ellas y tener en cuenta varios aspectos:

- para empezar, la sintaxis de las funciones puede resultar más confusa para ciertas personas e incluso hacer que sean más difíciles de leer al estar escritas de forma menos natural,
- **`this`** funciona de un modo diferente: su valor no puede ser modificado dentro de la función (funciones puras),
- **no pueden ser utilizadas como constructores** del mismo modo que el resto de funciones. Si se utiliza la palabra reservada **`new`** se obtendrá un error. Estas funciones no tienen una propiedad **`prototype`** u otros métodos internos,
- **no disponen de una variable interna definida como `arguments`** como sí tienen otras funciones. Hay que recordar que **`arguments`** es un pseudo-array que permite obtener los argumentos de una función, algo útil en caso de que una función tenga un número indeterminado de argumentos.

2.10.4. Ámbito de variables (`let` y `var`)

Anteriormente se ha hablado de la diferencia entre **`var`** y **`let`**, aunque no se ha profundizado mucho en ella. Ya se sabe que limita el alcance de una variable al bloque que lo contiene, es decir, una zona del código encerrada entre llaves **`{}`**. Sin embargo, es posible que el siguiente comportamiento todavía cause confusión:

```
function saluda() {
    var nombre = "Feo";
    let apellido = "Mucho";
    console.log(saludo, nombre, apellido); // Hola Feo Mucho
}
let saludo = "Hola";
saluda();
console.log(nombre); // Uncaught ReferenceError: nombre is not defined
```

¿Por qué no puede acceder a la variable **`nombre`** si se ha declarado con **`var`**? ¿Y ahora?:

```
let saludo = "Hola";
if(true) {
    var nombre = "Feo";
    let apellido = "Mucho";
    console.log(saludo, nombre, apellido); // Hola Feo Mucho
}
console.log(nombre); // Feo
console.log(apellido); // Uncaught ReferenceError: apellido is not defined
```

Ahora sí que se puede acceder a la variable **`nombre`**, pero no se puede acceder a la variable **`apellido`**. Es posible pensar que no todos los bloques se comportan igual en *JavaScript*, pero la realidad es que en el primer ejemplo entra en juego otro concepto muy importante para entender el ámbito: el **contexto de ejecución**, es decir, en qué función se está ejecutando el código actual. El código que no está dentro de ninguna función se encuentra en el **contexto de ejecución global**. Cada vez que se ejecuta una función se genera un nuevo contexto de ejecución creando ,a su vez, un nuevo ámbito.

Para entenderlo mejor es útil pensar en los ámbitos como cajas que dentro pueden contener dos cosas: variables y otras cajas. Cada caja puede acceder a sus variables sin problemas. Además, también puede acceder a las variables de las cajas más grandes, las que la contienen, pero nunca puede acceder a las variables que hay en cajas más pequeñas, las que contiene.

En el siguiente ejemplo, el contexto de ejecución es global, por lo que en todos los bloques es posible acceder a la variable declarada con **`var`** en el bloque más interno, esto no sería posible con **`let`** ya que quedaría adscrito al bloque que la definió:

```
{
    {
        {
            var nombre = "Feo";
            console.log(nombre);
        }
        console.log(nombre);
    }
    console.log(nombre);
```

```
}
console.log(nombre);
```

En resumen, se dice que las variables declaradas con **`var`** tienen ámbito de función (*function scoped*) mientras que las variables declaradas con **`let`** tienen ámbito de bloque (*block scoped*).

Cuando el motor de ejecución se encuentra con una variable primero mira si existe en el ámbito (en la caja) actual. Si no la encuentra, la intenta buscar en el ámbito superior y así hasta que llega al ámbito global. Si aún así no lo encuentra, actúa en consecuencia dependiendo de si se está utilizando la variable como fuente de datos (aquí dará error) o como destino de una asignación (aquí creará una nueva variable global si no está activado el modo estricto o dará error si está activado).

2.10.5. Alcance de funciones: *call stack* y *scope chain*

El alcance determina desde donde se puede acceder a una variable o una función, es decir, donde nace y donde muere. El **alcance global** significa que cualquier variable o función global pueden ser invocada o accedida desde cualquier parte del código de la aplicación. En *JavaScript*, por defecto, **todas las variables y funciones que definimos tienen alcance global**.

Si se define una variable dentro de una función, el **alcance se conoce como local o alcance de función**, de modo que la variable vive mientras lo hace la función. Aquella función que se define dentro de una función (padre) es local a la función pero global para las funciones anidadas (hijas) a la que hemos definido la función (padre). Por esto, más que alcance de función, se le conoce como **alcance anidado**.

Y así sucesivamente, se puede definir funciones dentro de funciones con alcance anidado en el hijo que serán accesibles por el nieto, pero no por el padre.

```
var varGlobal = "Esta es una variable global.";
var funcionGlobal = function (alfa) {
  var varLocal = "Esta es una variable local con alcance anidado.";
  var funcionLocal = function () {
    var varLocal = "¡Hola Feo!";
    console.log(varLocal);
    console.log(alfa);
  };
  funcionLocal();
  console.log(varLocal);
};
funcionGlobal(2);
```

Ahora se verán varios ejemplos con bloques de funciones. En este primer ejemplo, todas las funciones están al mismo nivel, y se van llamando unas a otras:

```
var nombre = "Feo";
function tercera() {
    var c = "C ";
    var texto2 = c + nombre;
    console.log(texto2);
}
function segunda() {
    var b = "B ";
```

```
    tercera();
    var texto2 = b + nombre;
    console.log(texto2);
}
function primera() {
    var a = "A ";
    segunda();
    var texto = a + nombre;
    console.log(texto);
}
primera();
```

La función **tercera** no tiene acceso a las variables **a**, **b** o **texto** porque no están definidas en un ámbito superior. Hay que distinguir muy bien entre el ***call stack*** (la pila de llamadas a funciones) que en este caso sería **global** → **primera** → **segunda** → **tercera**, de la ***scope chain*** (la cadena de ámbitos), que se construye en función de dónde está escrito el código. Este concepto también se conoce como ***lexical scope***. En el ejemplo de arriba, la función **tercera** tiene acceso a sus variables (**c** y **texto2**) y a las del ámbito global que es donde está escrita la función tercera.

Otro ejemplo:

```
var a = "A";
function primera() {
    var b = "B";
    function segunda() {
        var c = "C";
        function tercera() {
            var d = "D";
            console.log(a, b, c, d);
        }
        tercera();
    }
    segunda();
}
primera();
```

En este caso la función **tercera** tiene acceso a todas las variables porque está escrita dentro de **segunda**, que a su vez está escrita dentro de **primera**, que pertenece al **ámbito global**.

Un último ejemplo:

```
var a = "A";
function primera() {
    var b = "B";
    segunda();
    function segunda() {
        var c = "C"; tercera();
    }
}
function tercera() {
```

```
    var d = "D";
    console.log(a,b,c,d); //error
}
primera();
```

En este ejemplo, la función **tercera** de nuevo sólo tiene acceso a sus variables y a las globales. No importa que haya sido llamada desde **segunda**, importa que está declarada directamente sobre el ámbito global.

2.10.6. El pseudoarray `arguments`

Además de los parámetros declarados, cada función recibe siempre dos parámetros adiciones: **this** y **arguments**. El parámetro **this** será tratado con detalle en unidades posteriores.

El parámetro adicional **arguments** da acceso a todos los argumentos recibidos mediante la invocación de la función, incluso los argumentos que sobraron y no se asignaron a parámetros. Esto permite escribir funciones que tratan un número indeterminado de parámetros.

Estos datos se almacenan en una estructura similar a un **array**, aunque realmente no lo sea. Pese a ello, sí que tiene la propiedad **length** para obtener el número de parámetros y podemos acceder a cada elemento mediante la notación **arguments[x]**, pero carece del resto de métodos que ofrecen los **arrays**.

Por ejemplo, se puede crear una función que sume un número indeterminado de parámetros:

```
var suma = function() {
    var i, s=0;
    for (i=0; i < arguments.length; i+=1) {
        s += arguments[i];
    }
    return s;
};
console.log(suma(1, 2, 3, 4, 5)); // 15
```

2.11. *JavaScript Object Notation* (JSON)

Es un formato texto con el que se puede representar la estructura de elementos. Convierte variables, *arrays* y objetos en cadenas de texto. Por su simplicidad su uso no se ha quedado sólo en *JavaScript*, sino que se ha convertido en un estándar de facto para intercambiar datos entre sistemas.

Un formato del objeto **JSON** es un par de llaves que rodean cero o más parejas de **clave:valor** separados por comas, donde cada clave se considera como un propiedad/método. Se estructura de la siguiente manera:

```
var nadie = {};
var persona = {
    nombre: "Feo",
    apellido1: "De Verdad",
    direccion: {  // Puede haber JSON anidados.
        calle: "La de siempre",
        numero: 5,
        telefono: 123456789,
    }
}; // Todo objeto JSON finaliza con un punto y coma ( ; ).
```

La propiedad y el valor se separan con dos puntos, y cada una de las propiedades con una coma. Para recuperar un campo, además de la notación de punto, se puede acceder a cualquier propiedad usando la notación de corchetes:

```
var nom = persona.nombre;
var ape1 = persona["apellido1"];
var nombreCompleto = persona.nombre + persona.apellido1;
var calle = persona.["direccion"]["calle"];
var calle2 = persona.direccion.calle; // Segunda forma de acceso.
```

Es posible declarar una función que devuelva un objeto (a modo de constructor):

```
function creaPersona(nom, ape1) {
    return {
        nombre : nom,
        apellido1 : ape1,
        direccion: {
            calle: "La de siempre",
            numero: 5
            telefono: 123456789
        },
        getNombreCompleto : function() {
            return this.nombre + " " + this.apellido1;
        },
        saluda: function(persona) {
            if (typeof persona.getNombreCompleto !== "undefined") {
                return "Hola " + persona.getNombreCompleto();
            } else {
                return "Hola colega";
```

```
            }
        }
    };
};
var persona = creaPersona("Feo", "De Verdad"),
persona2 = creaPersona("Bruce", "Wayne");
persona.saluda(persona2); // Hola Bruce Wayne.
persona.saluda({}); // Hola colega.
```

Una propiedad de **JSON** también puede contener funciones, ya que son objetos y se pueden almacenar sin problemas.

Más información sobre el formato aquí .

2.11.1. Conversión

Para convertir un objeto a texto siguiendo el formato **JSON** se utiliza:

```
textoJSON = JSON.stringify(objeto);
```

Y para convertir una cadena de texto en **JSON** a un objeto (operación inversa a la anterior):

```
var objeto = JSON.parse(textoJSON);
```

Por ejemplo:

```
let miArray = new Array();
miArray[0] = "Feo";
miArray[1] = "De Verdad";
let textoJSON = JSON.stringify(miArray);
let arrayReconstruido = JSON.parse(textoJSON);
console.log(miArray); // Array original.
console.log(textoJSON); // Una cadena de texto en formato JSON.
console.log(arrayReconstruido); // Array reconstruido.
```

2.11.2. Encadenamiento opcional (*optional chaining*)

Cuando se accede a una propiedad de un objeto **JSON** y no existe (o no contiene valor) por defecto se devuelve el valor **undefined**. Esto puede acarrear serios problemas si se espera un valor de forma obligatoria para que el programa siga avanzando. Es posible establecer una comprobación del valor de esa propiedad antes de continuar con la ejecución del programa. Desde ESCMAScript 2015 se ha introducido un operador que permite evitar estas comprobaciones tan tediosas y se utiliza sobre todo en la programación reactiva, pero es interesante tenerlo presente:

```
var calle3 = persona.direccion?.calle; // Sólo accede a calle si dirección existe.
```

2.12. Operador *spread* (...)

Se trata de un operador integrado en **ECMAScript6** y se utiliza para distribuir los elementos de un objeto iterable (cadena de texto, array o cualquier cosa que se pueda recorrer) dentro de otro contenedor (también iterable). Es algo muy práctico ya que permite realizar operaciones sobre objetos iterables sin necesidad de conocer su prototipo.

Por ejemplo permiten copiar arrays:

```
let animales = ['perro', 'gato', 'feo'];
console.log(animales); // Muestra 'perro', 'gato', 'feo'.
let copiaDeAnimales = [...animales];
console.log(copiaDeAnimales); // Muestra 'perro', 'gato', 'feo'.
```

Añadir elementos a un objeto:

```
var persona = {
    nombre: "Feo",
    apellido1: "De Verdad",
}
var persona2 = { ...persona, 'apellido2': "De la buena"}
console.log(persona2);
```

Concatenar arrays:

```
let numeros = [1, 2, 3];
let meses = ['enero', 'febrero', 'marzo'];
console.log([...numeros, ...meses]);
```

Pasar parámetros a funciones:

```
let numeros = [1, 2, 3];
const sumaNumeros = (a, b, c) => {
    console.log(a + b + c);
}
sumaNumeros(...numeros); // Muestra 6.
```

Conviene familiarizarse con su uso ya que simplificará el código escrito. Amplia la información sobre este operador aquí .

2.13. Desestructuración

Es otra de las novedades de **ECMAScript6** que facilita la escritura de código es que es posible asignar los valores de un objeto a variables de un modo rápido.

Por ejemplo:

```
let persona = {
    nombre: "Feo",
    apellido1: "De Verdad",
    apellido2: "De La Buena"
}
let {nombre, apellido1} = persona;
console.log(nombre) // Muestra Feo.
console.log(apellido1); // Muestra De Verdad.
```

De este modo no es necesario usar la nomenclatura de punto para el acceso a valores de un objeto iterable (ni tampoco los corchetes) y siempre funcionará si las variables de destino se llaman igual que las propiedades del objeto iterable. Si es necesario cambiar el nombre a los contenedores de destino es posible realizarlo de este modo:

```
let {nombre: nom, apellido1: cognom1, malnom = "Ninguno"} = persona;
console.log(nom) // Muestra Feo.
console.log(cognom1); // Muestra De Verdad.
console.log(malnom); // Muestra Ninguno.
```

Además, permite utilizar la definición de parámetros por defecto. En este caso la variable **malnom** no existe en el objeto por lo que se le asigna el valor **"Ninguno"**. Si el objeto tuviera una propiedad con ese nombre le asignaría su valor y no el especificado por defecto.

Todo esto también funciona con **arrays**, aunque la asignación a los contenedores de destino se realizará de forma posicional:

```
let sorpresa = ["Luke", "yo" , "soy", "tu", "padre"];
let [nombre, pronombre, verbo] = sorpresa;
console.log(nombre); // Muestra Luke.
console.log(pronombre); // Muestra yo.
console.log(verbo); // Muestra soy.
```

Se puede asignar el resto del array a otro objeto iterable de destino:

```
let sorpresa = ["Luke", "yo" , "soy", "tu", "padre"];
let [nombre, ...resto] = sorpresa;
console.log(nombre); // Muestra Luke.
console.log(resto); // Muestra yo soy tu padre.
```

Al igual que en el caso anterior, la desestructuración resulta muy útil y su manejo simplifica el código, y eso es siempre una buena idea.

Más información aquí .

2.14. Objetos predefinidos (adelanto UT07)

JavaScript es un lenguaje que, de forma nativa, posee gran cantidad de funciones y objetos predefinidos que son útiles para realizar un código más eficiente y claro. El uso de estos objetos reducirá el tiempo de desarrollo y también el de prueba de las aplicaciones.

2.14.1. Funciones predefinidas

- **Tratamiento numérico**

Todos los números en *JavaScript* se almacenan como números de punto flotante de 64 bits. Para crear variables numéricas se asigna el valor a la variable independientemente de su formato:

```
var diez = 10; // Entero.
var pi = 3.14; // Real.
```

Si se necesita redondear una cifra a un número determinado de decimales usaremos el método **`toFixed(dígitos)`**, el cual devuelve la cifra original con tantos decimales como los indicados por el parámetro:

```
var pi = 3.14159265;
console.log(pi.toFixed(0)); //3
console.log(pi.toFixed(2)); //3.14
console.log(pi.toFixed(4)); //3.1416
```

Para convertir una cadena a un número se utiliza la función global **`parseInt(cadena [, base])`**, donde la base por defecto es 10, es decir, decimal. Del mismo modo para pasar a un número real usaremos la función global **`parseFloat(cadena [, base])`**.

```
var cadena = "3.14";
var pi = parseFloat(cadena, 10);
var tres = parseInt(pi, 10);
```

JavaScript emplea el valor **`NaN`** (que significa *Not A Number*) para indicar un valor numérico no definido, por ejemplo, la división 0/0 o al convertir un texto que no coincide con ningún número.

```
var numero1 = 0;
var numero2 = 0;
console.log(numero1 / numero2); // NaN
console.log(parseInt("tres")); // NaN
```

Para averiguar si una variable no es un número, se usa la función **`isNaN(valor)`**:

```
var miNumero = "tres";
if (isNaN(miNumero)) {
  console.log("¡No es un número!");
}
```

- **Función `eval`**

`Eval` es una función que recibe una cadena y la interpreta como código *JavaScript*. Dado que *JavaScript* admite expresiones numéricas, se puede usar **`eval`** para calcular su resultado. Es una función muy útil ya que se puede construir código dinámicamente mediante una cadena. Debe utilizarse únicamente en situaciones que lo requiera, ya que una cadena interpretada por **`eval`** que esté formada maliciosamente puede causar un agujero de seguridad.

```
let x = 3;
let y = 2;
let a = eval("2+3");
let b = eval("x*y");
eval("console.log('a vale '+a+' b vale '+b)");
```

2.14.2. Objeto predefinido `String`

Cuando se crea una cadena automáticamente se convierte en un objeto **`String`** con sus propiedades y métodos predefinidos. Esto es útil porque ayuda a realizar múltiples operaciones con cadenas de una manera rápida y sencilla. Hay que tener en cuenta que los métodos de manipulación de cadenas de *JavaScript* no modifican al objeto actual, sino que **devuelven el objeto resultante de aplicar la modificación**. Si se necesita que la modificación se aplique sobre la misma cadena que estamos trabajando, se hará algo así:

```
cadena=cadena.metodoQueModificaCadena();
```

En este objeto hay una propiedad muy importante llamada **`length`** que indica cuantos elementos (caracteres) tiene la cadena. Además existen una serie de métodos muy útiles:

- **`toLowerCase()/toUpperCase()`**, devuelve la cadena convertida a minúsculas/mayúsculas,
- **`concat(cadena)`** devuelve el objeto con el valor de cadena concatenado al final,
- **`charAt(posición)`**, devuelve el carácter que se encuentre en la posición solicitada. Se debe tener en cuenta que las posiciones comienzan a contar desde cero,
- **`indexOf(texto, [indice])`**, devuelve la primera posición donde se encuentra el texto buscado, empezando a buscar desde la posición **`indice`**. Si no se indica, se toma por defecto el valor 0,
- **`lastIndexOf(texto, [indice])`**, como la anterior, busca hacia atrás la primera ocurrencia del texto buscado. Si no se indica el valor de **`indice`**, se busca desde el final,
- **`replace(texto1,texto2)`**, busca la cadena **`texto1`** y las reemplaza por **`texto2`**,
- **`split(caracter, [trozos])`**, separa la cadena mediante un separador. **`Trozos`** indica el máximo de separaciones. Si no se indica, se harán todas las separaciones posibles,
- **`substring(inicio, [fin])`**, devuelve la subcadena comprendida entre la posición **`inicio`** y **`fin`**. Si **`fin`** no se indica, se toma como valor el final de la cadena,

```
let cad = "Feo:De Verdad:654875214";
let tfo;
cad = cad.toUpperCase();
console.log(cad);
splitTodosCampos = cad.split(":");
split1Campo = cad.split(":", 1);
console.log(splitTodosCampos);
console.log(split1Campo);
tfo = splitTodosCampos[2];
```

```
tfo = tfo.replace("2", "9"); // Cambia en el teléfono los números 3 por 9s.
console.log(tfo);
console.log(tfo.charAt(4)); // Muestra el quinto número del teléfono.
```

2.14.3. Objeto predefinido `Date`

Para crear una fecha se usa el objeto **`Date`**. Si se usa el constructor vacío se obtendrá la fecha actual. Pasándole un valor que representa el *timestamp Epoch* (desde el 1/1/1970) o pasándole al constructor del día (1-31), mes (0-11) y año, se crea un objeto con una fecha predeterminada.

```
var fecha = new Date();
console.log(fecha);
var nochevieja = new Date(2050, 11, 31);
console.log(nochevieja);
```

Si además es necesario indicar la hora se hace mediante tres parámetros más:

```
var cenaNochevieja = new Date(2050, 11, 31, 22, 30, 0);
console.log(cenaNochevieja);
```

Una vez creado un objeto **`Date`** , existen métodos para realizar operaciones:

- **`getFullYear()`** devuelve el año de la fecha con cuatro dígitos,
- **`getMonth()`** número del mes del año (de 0 a 11),
- **`getDate()`** número de día del mes.

Es posible añadir o sustraer unidades de tiempo al objeto fecha utilizando operadores aritméticos **`+`** o **`-`** :

```
var cenaNochevieja = new Date(2050, 11, 31, 22, 30, 0);
var anyo = cenaNochevieja.getFullYear();
var mes = cenaNochevieja.getMonth();
var diaMes = cenaNochevieja.getDate();
var resaca = new Date(cenaNochevieja.setDate(diaMes + 1));
console.log(resaca);
```

Más información aquí .

Para comparar fechas se usan los operadores **`<`** , **`>`** o la comparación de igualdad/identidad.

```
var cenaPreNochevieja = new Date(2050, 11, 30, 22, 30, 0);
var cenaNochevieja = new Date(2050, 11, 31, 22, 30, 0);
var cenaNochevieja2 = new Date(2050, 11, 31, 22, 30, 0);
console.log(cenaPreNochevieja < cenaNochevieja); // true
console.log(cenaNochevieja == cenaNochevieja2); // false
console.log(cenaNochevieja === cenaNochevieja2); // false
console.log(cenaNochevieja.getTime() == cenaNochevieja2.getTime()); // true
console.log(cenaNochevieja.getTime() === cenaNochevieja2.getTime()); // true
```

Trabajar con fechas siempre es problemático dado que el propio lenguaje no ofrece métodos para realizar cálculos sobre fechas o realizar consultas utilizando el lenguaje natural. Una biblioteca muy completa es ***Datejs*** de la que se puede obtener más información aquí .

2.14.4. Objeto predefinido Math

El objeto **`Math`** es un objeto que contiene una colección de métodos que ayudarán a realizar cualquier operación que no sea operaciones aritméticas básicas:

- potencia **`Math.pow(base,exp)`**,
- raíz cuadrada **`Math.sqrt(num)`**,
- redondear **`Math.round(num)`**, **`Math.ceil(num)`**, **`Math.floor(num)`**,
- obtener el mayor **`Math.max(num1, num2,…)`**,
- obtener el menor **`Math.min(num1, num2, …)`**,
- un número aleatorio en 0 y 1 **`Math.random()`**,
- el número pi **`Math.PI`**

y un largo etcétera. Más información aquí.

2.14.5. Objeto predefinido `Array`

Es un tipo predefinido que, a diferencia de otros lenguajes, es un objeto. Del mismo modo que los tipos básicos, se puede crear de la siguiente manera:

```
var cosas = new Array();
var tresTipos = new Array(11, "Feo", true);
var tresTiposDos = [11, "hola", true];
var longitud = tresTipos.length; // 3
var once = tresTipos[0];
```

Se observa que en *JavaScript* los *arrays* pueden **contener tipos diferentes**, que el **primer elemento es el cero** y que se obtiene su longitud mediante la propiedad **`length`** .

Se pueden añadir elementos en la posiciones que se necesite, aunque se recomienda añadir los elementos en posiciones secuenciales. Si se accede a un elemento que no contiene ningún dato se obtendrá **`undefined`**.

```
tresTipos[3] = 15;
tresTipos[tresTipos.length] = "Bruce";
var longitud2 = tresTipos.length; // 5
tresTipos[8] = "Wayne";
var longitud3 = tresTipos.length; // 9
var nada = tresTipos[7]; // undefined
```

Al trabajar con una cadena de texto una operación que se utiliza mucho es dividirla en trozos a partir de un separador. Al utilizar el método **`String.split(separador)`** devolverá un *array* de cadenas con cada uno de los trozos.

```
var frase = "Yo soy Batman.";
var arrayPalabras = frase.split(" ");
var yo = arrayPalabras[0];
var soy = arrayPalabras[1];
var Batman = arrayPalabras[2];
```

Algunos de los métodos más importantes del objeto *array*:

- **`join([separador])`**, devuelve una cadena con todos los elementos del *array* separados por el texto que se incluya en **`separador`**,
- **`push(elemento)`**, añade al final del array **`elemento`**,

- **`pop()`**, devuelve y elimina el último elemento del *array,*
- **`reverse()`**, invierte el orden de los elementos de un *array,*
- **`sort()`**, ordena los elementos de un *array* alfabéticamente,
- **`slice(inicio, [final])`**, devuelve los elementos de un *array* comprendidos entre **`inicio`** y **`final`**. Si no se indica **`final`**, se toma hasta el último elemento del *array,*
- **`find()`**, devuelve la primera coincidencia del elemento que se busca.

En este enlace se encuentra una guía completa de todos los métodos de su prototipo: aquí .

- **Recorrer un `Array`**

JavaScript ofrece métodos para recorrer un *array* de forma rápida y sencilla. Los siguientes métodos aceptan una función *callback* como primer argumento e invocan dicha función para cada elemento del *array*. La función que se le pasa a los métodos reciben siempre tres parámetros:

- el **valor** del elemento del *array*,
- el **índice** del elemento y
- el propio ***array***.

La mayoría de las veces sólo es necesario utilizar el valor. Los métodos más utilizados son:

- **`forEach(función)`**, ejecuta la función para cada elemento del *array,*
- **`map(función)`**, ejecuta la función para cada elemento del *array*, y el nuevo valor se inserta como un elemento del nuevo *array* que devuelve,
- **`every(función)`**, **`true`** si la función se cumple para todos los valores, **`false`** en caso contrario,
- **`some(función)`**, **`true`** si la función se cumple para al menos un valor, **`false`** si no se cumple para ninguno de los elementos,
- **`filter(función)`**, devuelve un nuevo *array* con los elementos que cumplen la función,
- **`reduce(función)`**, ejecuta la función para un acumulador y cada valor del *array* (de inicio a fin) se reduce a un único valor.

Por ejemplo, un uso de **`map()`**:

```
var heroes = ["Batman", "Superman", "Ironman", "Thor"];
function mayus(valor, indice, array) {
  return valor.toUpperCase();
}
var heroesMayus = heroes.map(mayus);
console.log(heroesMayus); // ["BATMAN", "SUPERMAN", "IRONMAN", "THOR"]
```

Mostrar todos los elementos del *array* con **`forEach()`**:

```
var heroes = ["Batman", "Superman", "Ironman", "Thor"];
heroes.forEach(function (valor, indice) {
  console.log("[", indice, "]=", valor);
});
```

Comprobar si todos los elementos de un *array* son cadenas con el método **`every()`**. Para ello se crea una función **`esCadena`** que se usará como *callback*:

```
function esCadena(valor, indice, array) {
  return typeof valor === "string";
}
console.log(frutas.every(esCadena)); // true
```

Para un *array* con datos mezclados con textos y números, es posible filtrar los elementos que son cadenas con **`filter()`**:

```
var mezcladillo = [1, "dos", 3, "cuatro", 5, "seis"];
console.log(mezcladillo.filter(esCadena)); // ["dos", "cuatro", "seis"]
```

Finalmente, mediante el método **`reduce()`** se puede realizar un cálculo sobre los elementos del *array*. Por ejemplo, contar cuantas veces aparece una ocurrencia dentro de un *array* o sumar sus elementos. Para ello la función recibe un parámetro extra que funciona a modo de acumulador, además de los parámetros valor, indice y *array* de costumbre:

```
var numeros = [1, 3, 5, 7, 9];
var suma = numeros.reduce(function (acumulador, valor, indice, array) {
  return acumulador + valor;
});
```

En el primer paso, como no hay valor inicial, se pasan el primer y el segundo elemento del *array* (los valores 1 y 3). En siguientes iteraciones, el valor **`acumulador`** es lo que devuelve el código, y **`valor`** es el siguiente elemento del *array*. De este modo, estamos cogiendo el valor actual y sumándoselo al valor anterior (el total acumulado).

Aunque parezcan el mismo método, **`forEach()`** y **`map()`** tienen una diferencia fundamental y es que el segundo devuelve un valor tras la ejecución de su *callback* pero no así el primero:

```
var feo = [1, 2, 3, 4],
  nuevo = [];
nuevo = feo.forEach((valor) => valor * 5);
console.log(nuevo); //undefined
nuevo = feo.map((valor) => valor * 5);
console.log(nuevo); // [5, 10, 15, 20]
```

Uso de arrays en programación reactiva

Cuando se trabaja con **`React`** hay que tener presente que los componentes y los estados (que se verá en lo sucesivo) deben trabajar como funciones puras: no deben modificar directamente los parámetros recibidos. Por eso, en **`React`**, es habitual clonar los parámetros para poder modificarlos y devolverlos una vez producido el cambio. Hay que tener muy presente este hecho.

```
const feos = ["Rodrigo", "Juan", "Artura", "Javier"];
// Recorrer un objeto (no se modifica pero hay que utilizar map).
const feos2 = feos.map((feo) => {
  console.log(feo);
  return feo;
});
console.log(feos2);
// Añadir un nuevo valor al objeto (uso de spread operator).
const nuevoFeo = "Juan Carlos";
```

```
const feos3 = [...feos, nuevoFeo];
console.log(feos3);
// Eliminar un elemento del objeto (filter).
const feos4 = feos3.filter((feo) => {
  return feo !== nuevoFeo;
});
console.log(feos4);
// Actualizar un elemento del objeto.
const nuevoValor = "Arturo";
const valorCambiar = "Artura";
const feos5 = feos.map((feo) => {
  if (feo === valorCambiar) {
    return nuevoValor;
  } else {
    return feo;
  }
});
console.log(feos5);
```

2.15. Componentes en React

Los componentes permiten separar interfaces de usuario en piezas independientes y reutilizables. Internamente, para *JavaScript*, un componente no es más que una función que recibe un objeto, que se denomina **props**, y retorna un elemento de **React** que describe qué va a aparecer en la pantalla. Se utilizan para indicarle a la aplicación qué es lo que es necesario ver en pantalla y que cuando los datos cambien sólo actualice el componente y no el resto de la aplicación.

Quizás la parte más complicada de esta dinámica es la de dividir la aplicación web en componentes, aunque es una práctica que se adquiere con el tiempo. Un componente debe tratarse de forma independiente aunque debe interactuar con el resto de ellos que conforman la aplicación. Será necesario comunicarse entre sí y esta tarea se realiza a través de sus propiedades (**props**).

Es importante tener en cuenta que los componentes de **React** deben empezar con una mayúscula. De esta forma es como se diferencian los elementos nativos de **HTML** de los componentes creados de forma pormenorizada. Si no se hace así, el preprocesador del código lo interpreta como si fuese una cadena (**string**) y no como si fuese una función (componente) y no lo dibujará (renderizará) de la manera esperada.

Un componente en **React** está compuesto por dos partes: la parte que contiene la **lógica de funcionamiento** (que se escribe en *JavaScript)*, y la **parte que devuelve** (a través de **return**) que está escrita en **JSX** (que representa el código que el navegador mostrará en pantalla).

```
function ComponenteFeo(props) {
  // Lógica del componente en código JavaScript.
  return (
    // Vista del componente que será pintada en el navegador escrita en JSX.
  );
}
```

Un componente sólo puede dibujar (renderizar) el contenido **JSX** encerrado dentro de una etiqueta ** ** ya que posee la limitación de estar obligado a devolver una etiqueta en cada dibujado:

```
function Undiv(props){
 return(
    <div>
      Estoy devolviendo un sólo div.
    </div>
    )
}
```

Si es necesario devolver dos etiquetas en la raíz del dibujado, se utilizan los **React Fragments** o encerrando todas las etiquetas en la raíz entre dos etiquetas vacías **<>**:

```
function Dosdivs(props) {
  return (
    <React.Fragments>
      <div>Estoy devolviendo el primer div.</div>
      <div>Y ahora es segundo div.</div>
    </React.Fragments>
```

```
  );
}
```

2.15.1. Tipos de componentes

Existen dos tipos de componentes:

- **Componentes de clases** (*Class components*), utiliza la programación orientada a objetos (POO) para definir un componente. Para que una clase se considere componente en **React** debe heredar de la clase **React.Component** y debe contar con un método **render** el cual debe devolver elementos de **React** (otro componente, código **JSX** o datos, por ejemplo).

  ```
  // Componentes de clases y ECMAScript 6.
  class conponenteFeo extends React.Component {
    render() {
      return (
        <div className='componente'>
          <h2>Hijo 1</h2>
          <p>Contador Feo 1</p>
          <p>1</p>
        </div>
      );
    }
  }
  ```

  La principal ventaja de usar *class components* es que es posible utilizar el **estado de la información** para cambiar su estructura. Esta característica, que se estudiará con posterioridad, era exclusiva de este tipo de componentes hasta el advenimiento de los **Hooks** en la versión 16.8.0 de **React**. Esta adición permite prescindir de las clases por completo y utilizar funciones para el control del estado de la información.

- **Componentes funcionales** (*Functional Components*), es la forma más sencilla con la que podemos definir un componente en **React** ya que se utilizan funciones tal y como se haría en *JavaScript*. Además, es la forma de crear estos elementos que más se utiliza entre los programadores de **React** mientras que, progresivamente, se abandona en uso de los componentes de clase.

  Para que una función se convierta en *functional component* debe aceptar un sólo argumento (**props**) o ninguno y debe devolver código escrito en **JSX**.

  ```
  //Componentes funcionales.
  function ComponenteFeo(props) {
    return (
      <div className='padre'>
        <h1>Componente Padre</h1>
        <div className='componentes'>
          <div className='componente'>
            <h2>Hijo 1</h2>
            <p>Contador Feo 1</p>
            <p>1</p>
          </div>
  ```

```
        <div className='componente'>
          <h2>Hijo 2</h2>
          <p>Contador Feo 2</p>
          <p>1</p>
        </div>
      </div>
    </div>
  );
}
```

2.15.2. Propiedades en un componente (`props`)

Cuando se utiliza un componente es posible parametrizarlo a través de las propiedades que se especifican durante su declaración. No hay límite en cuanto a la cantidad de propiedades que se le pueden dar a un componente. Eso sí, las propiedades son sólo de lectura, es decir, los componentes son **funciones puras** (no modifican sus parámetros).

Así funciona un componente utilizando propiedades:

```
function Feo2(props) {
  return (
    <div className='componente'>
      <h2>{props.titulo}</h2>
      <p>{props.subtitulo}</p>
      <p>{props.nota}</p>
    </div>
  );
}
```

Y así su declaración dentro de otro componente:

```
<Feo2 titulo=”El bueno, el malo…” subtitulo=”...y el feo.” nota=”90/100” />
```

Cada componente dispone de una propiedad especial que es **`children`**. Hay que recordar que un componente en **`React`** se maneja como si una etiqueta de **`HTML`** se tratase, por lo que **`children`** hará referencia al contenido introducido entre las etiquetas y no al pasado como propiedades. Si se utiliza el ejemplo anterior:

```
function Feo2(props) {
  return (
    <div className='componente'>
      <h2>{props.titulo}</h2>
      <p>{props.children}</p>
      <p>{props.nota}</p>
    </div>
  );
}
```

Y esta sería su construcción:

```
<Feo2 titulo=”El bueno, el malo…” nota=”90/100”> … y el feo </Feo2>
```

2.15.3. JSX

Para escribir componentes se utiliza la sintaxis **JSX** (*JavaScript* **XML**). Esta es una extensión a *JavaScript* que nos permite escribir **HTML** sin necesidad de representarlo como un **string** (encerrándolo entre comillas). En realidad es un preprocesador de código que, al ser dibujado (render), se traduce en código **HTML**.

```
function App() {
  return <h1>Hola Mundo</h1>;
}
```

JSX facilita la unión entre la lógica y la estructura de la aplicación:

```
React.createElement(type, [props], [...children]);
```

de modo que el ejemplo anterior transformado a **ECMAScript6** por un preprocesador se transforma en:

```
function App() {
  return React.createElement("h1", null, "Hola Mundo");
}
```

y que una vez realizado el render, queda así:

```
<h1>Hola Mundo</h1>
```

Por supuesto, es posible crear estructuras más complejas utilizando etiquetas **HTML** anidadas en **JSX**:

```
function App() {
  return (
    <div>
      <h1>Bienvenido Feo</h1>
      <button>Unirse a la comunidad</button>
    </div>
  );
}
```

Su traducción a **ECMAScript6** sería:

```
function App() {
  return React.createElement(
    "div",
    null,
    React.createElement("h1", null, "Bienvenido Feo"),
    React.createElement("button", null, "Unirse a la comunidad")
  );
}
```

Las etiquetas **<h1>** y **<button>** también son transformadas a una llamada a la función **React.createElement(...)** y son pasados como parámetros en la llamada a la función que crea el ** ** que los contiene.

También se pueden utilizar componentes de **React** para describir la interfaz de usuario.

```
function Saludo() {
 return (
   <h1>¡Hola Feo!</h1>
 );
}

function App() {
 return (
   <Saludo />
 );
}
```

Como se aprecia en estos sencillos ejemplos, sin la utilización de **JSX** el código se vuelve pesado de escribir y difícil de mantener, por lo que aprender **JSX** es un tiempo que retorna su inversión en un breve lapso de tiempo.

Expresiones *JavaScript* dentro de JSX

Se pueden incrustar expresiones dentro de **JSX** encerrándolas entre llaves:

```
const nombre = 'Feo Muy Feo';
const despedida = <h1>¡Adiós, {nombre}!</h1>;
```

y esto al dibujarlo en **HTML** será:

```
<h1>¡Adiós, Feo Muy Feo!</h1>
```

Dibujado condicional

Otra expresión bastante útil es la que permite dibujar una sección de una aplicación dependiendo de una condición. Un problema con este tipo de expresiones es que no permite implementar una lógica de tipo **`if-else`** nativa de *JavaScript*. En el caso de ser necesario este tipo de lógicas es posible utilizar el operador ternario:

```
function Viernes(props) {
  let esViernes = false;
  return(
    <div>
      <h1> Bienvenido </h1>
      <h1>{esViernes ? "¡Es viernes!" : "Otro día más..."}</h1>
    </div>
  );
}
```

que al dibujarlo (renderizarlo) quedaría:

```
<div>
  <h1> Bienvenido </h1>
  <h2>Otro día más...</h2>
</div>
```

Valores Ignorados

Las etiquetas que contengan valores booleanos (ya sean **`true`** o **`false`**), **`null`** y **`undefined`** serán dibujados sin ningún contenido dentro.

```
<p>{true}</p>
<p>{false}</p>
<p>{true && true}</p>
<p>{null}</p>
<p>{undefined}</p>
<p></p>
```

Todos los ejemplos anteriores transpilan a un párrafo vacío.

Si bien es posible escribir aplicaciones en **`React`** sin utilizar **JSX**, el uso de esta extensión permite agilizar la escritura del código. Las mejoras que ofrece utilizarlo no sólo se quedan en la simplicidad y claridad del código, sino que también permite más productividad y rapidez en el desarrollo de aplicaciones con **`React`**.

Aunque **JSX** es bastante simple de entender, tiene particularidades que es necesario conocer:

- hay ciertos atributos en **`HTML`** que son los mismos que palabras reservadas de *JavaScript* por lo que hay que cambiarlos en **JSX**. Por ejemplo **`class`** por **`className`** o **`for`** por **`htmlFor`**,
- hay que tener en cuenta que la sintaxis de **JSX** es más cercana a la sintaxis de *JavaScript* que de **`HTML`** y comparten la misma convención al momento de nombrar funciones o variables. Recuerda usar **`camelCase`**,
- todas las etiquetas se deben cerrar ya sea con una etiqueta de cierre **`<div></div>`** o por auto cerrado **`<img />`**,
- sólo permite devolver una única etiqueta, por lo que todas ellas deben estar envueltas dentro de una (aunque esto es más una limitación de componente que de **JSX**):

  ```
  <div>
    <h1>Hello Feo!</h1>
    <p>from Petrer</p>
  </div>
  ```

- se usan las llaves (**`{ }`**) para insertar expresiones de *JavaScript* en código **JSX**,
- se desaconseja utilizar el estilo en línea a través de la etiqueta **`style`**. **JSX** no interpreta de forma directa código **`CSS`**, sino que habrá que utilizar *Camel Case* para los atributos compuestos de **`CSS`**, como por ejemplo **`backgrundColor`**. Además este código debe estar encerrado entre dobles llaves **`{{ }}`**. Evita utilizar estilo en línea ya que siempre debe manejarse a través de las clases.

Teniendo en cuenta estas particularidades, aprender esta extensión de *JavaScript* no resulta una tarea difícil.