Tipos de datos

En Javascript, al igual que en la mayoría de los lenguajes de programación, al declarar una variable y guardar su contenido, también le estamos asignando un **tipo de dato**, ya sea de forma implícita o explícita. El **tipo de dato** no es más que la naturaleza de su contenido: contenido numérico, contenido de texto, etc...

¿Qué tipos de lenguajes existen?

A grandes rasgos, nos podemos encontrar con dos tipos de lenguajes de programación:

- **Lenguajes estáticos**: Cuando creamos una variable, debemos indicar el **tipo de dato** del valor que va a contener. En consecuencia, el valor asignado finalmente, siempre deberá ser del tipo de dato que hemos indicado (*si definimos que es un número debe ser un número, si definimos que es un texto debe ser un texto, etc...*).
- **Lenguajes dinámicos**: Cuando creamos una variable, no es necesario indicarle el tipo de dato que va a contener. El lenguaje de programación se encargará de deducir el tipo de dato (*dependiendo del valor que le hayamos asignado*).

En el caso de los **lenguajes dinámicos**, realmente el tipo de dato se asocia al valor (*en lugar de a la variable*). De esta forma, es mucho más fácil entender que a lo largo del programa, dicha variable puede «cambiar» a tipos de datos diferentes, ya que la restricción del tipo de dato está asociada al valor y no a la variable en sí. No obstante, para simplificar, en los primeros temas siempre hablaremos de variables y sus tipos de datos respectivos.

Javascript pertenece a los **lenguajes dinámicos**, ya que automáticamente detecta de que tipo de dato se trata en cada caso, dependiendo del contenido que le hemos asignado a la variable.

Para algunos desarrolladores —*sobre todo, nóveles*— esto les resulta una ventaja, ya que es mucho más sencillo declarar variables sin tener que preocuparte del tipo de dato que necesitan. Sin embargo, para muchos otros desarrolladores —*generalmente, avanzados*— es una desventaja, ya que pierdes el control de la información almacenada y esto en muchas ocasiones puede desembocar en problemas o situaciones inesperadas.

En Javascript existen mecanismos para convertir o forzar los tipos de datos de las variables, sin embargo, muchos programadores prefieren declarar explícitamente los tipos de datos, ya que les aporta cierta confianza y seguridad. Este grupo de desarrolladores suelen optar por utilizar lenguajes como Typescript, que no es más que «varias capas de características añadidas» a Javascript.

En muchas ocasiones (*y de manera informal*) también se suele hacer referencia a **lenguajes tipados** (*tipado fuerte, o fuertemente tipado*) o **lenguajes no tipados** (*tipado*

débil, debilmente tipado), para indicar si el lenguaje requiere indicar manualmente el tipo de dato de las variables o no, respectivamente.

¿Qué son los tipos de datos?

En Javascript disponemos de los siguientes tipos de datos:

Tipo de dato	Descripción	Ejemplo básico
number	Valor numérico (enteros, decimales, etc...)	`42`
string	Valor de texto (cadenas de texto, carácteres, etc...)	`'MZ'`
boolean	Valor booleano (valores verdadero o falso)	`true`
undefined	Valor sin definir (variable sin inicializar)	`undefined`
function	Función (función guardada en una variable)	`function() {}`
object	Objeto (estructura más compleja)	`{}`

Para empezar, nos centraremos en los tres primeros, denominados **tipos de datos primitivos**, y en los temas siguientes veremos detalles sobre los siguientes.

Para saber que tipo de dato tiene una variable, debemos observar que valor le hemos dado. Si es un valor numérico, será de tipo **number**. Si es un valor de texto, será de tipo **string**, si es verdadero o falso, será de tipo **booleano**. Veamos un ejemplo en el que identificaremos que tipo de dato tiene cada variable:

```
var s = "Hola, me llamo Manz"; // s, de string
var n = 42; // n, de número
var b = true; // b, de booleano
var u; // u, de undefined
```

Como se puede ver, en este ejemplo, es muy sencillo saber que tipos de datos tienen cada variable.

¿Qué tipo de dato tiene una variable?

Nos encontraremos que muchas veces no resulta tan sencillo saber que tipo de dato tiene una variable, o simplemente viene oculto porque el valor lo devuelve una función o alguna otra razón similar. Hay varias formas de saber que tipo de dato tiene una variable en Javascript:

Utilizando typeof()

Si tenemos dudas, podemos utilizar la función `typeof`, que nos devuelve el tipo de dato de la variable que le pasemos por parámetro. Veamos que nos devuelve `typeof()` sobre las variables del ejemplo anterior:

```
console.log(typeof s); // "string"
console.log(typeof n); // "number"
console.log(typeof b); // "boolean"
console.log(typeof u); // "undefined"
```

Como se puede ver, mediante la función **typeof** podremos determinar que tipo de dato se esconde en una variable. Observa también que la variable `u`, al haber sido declarada sin valor, Javascript le da un tipo de dato especial: **undefined** (*sin definir*).

La función `typeof()` solo sirve para variables con **tipos de datos** básicos o primitivos.

Utilizando constructor.name

Más adelante, nos encontraremos que en muchos casos, `typeof()` resulta insuficiente porque en tipos de datos más avanzados simplemente nos indica que son **objetos**. Con `constructor.name` podemos obtener el tipo de constructor que se utiliza, un concepto que veremos más adelante dentro del tema de clases. De momento, si lo necesitamos, podemos comprobarlo así:

```
console.log(s.constructor.name); // String
console.log(n.constructor.name); // Number
console.log(b.constructor.name); // Boolean
console.log(u.constructor.name); // ERROR, sólo funciona con variables
definidas
```

OJO: Sólo funciona en variables definidas (*no undefined*) y sólo en ECMAScript 6.

Que Javascript determine los **tipos de datos automáticamente** no quiere decir que debamos despreocuparnos por ello. En muchos casos, debemos conocer el tipo de dato de una variable e incluso necesitaremos convertirla a otros tipos de datos antes de usarla. Más adelante veremos formas de convertir entre tipos de datos.

Variables y constantes

En javascript es muy sencillo declarar y utilizar variables, pero aunque sea un procedimiento simple, hay que tener una serie de conceptos previos muy claros antes de continuar para evitar futuras confusiones, sobre todo si estamos acostumbrados a otros lenguajes más tradicionales.

Variables

En programación, las **variables** son espacios donde se puede guardar información y asociarla a un determinado nombre. De esta forma, cada vez que se consulte ese nombre posteriormente, te devolverá la información que contiene. La primera vez que se realiza este paso se suele llamar **inicializar una variable**.

En Javascript, si una variable no está inicializada, contendrá un valor especial: **undefined**, que significa que su valor no está definido aún, o lo que es lo mismo, que no contiene información:

```
var a; // Declaramos una variable "a", pero no le asociamos ningún
contenido.
var b = 0; // Declaramos una variable de nombre "b", y le asociamos el
número 0.

console.log(b); // Muestra 0 (el valor guardado en la variable "b")
console.log(a); // Muestra "undefined" (no hay valor guardado en la
variable "a")
```

Como se puede observar, hemos utilizado **console.log()** para consultar la información que contienen las variables indicadas.

OJO: Las mayúsculas y minúsculas en los nombres de las variables de Javascript **importan**. No es lo mismo una variable llamada **precio** que una variable llamada **Precio**, pueden contener valores diferentes.

Si tenemos que declarar muchas variables consecutivas, una buena práctica suele ser escribir sólo el primer **var** y separar por comas las diferentes variables con sus respectivos contenidos (*método 3*). Aunque se podría escribir todo en una misma línea (*método 2*), con el último método el código es mucho más fácil de leer:

```
// Método 1: Declaración de variables de forma independiente
var a = 3;
var c = 1;
var d = 2;

// Método 2: Declaración masiva de variables con el mismo var
var a = 3,
  c = 1,
  d = 2;
```

```
// Método 3: Igual al anterior, pero mejorando la legibilidad del
código
var a = 3,
  c = 1,
  d = 2;
```

Como su propio nombre indica, una **variable** puede variar su contenido, ya que aunque contenga una cierta información, se puede volver a cambiar. A esta acción ya no se le llama inicializar una variable, sino **declarar una variable** (*o más concretamente, redeclarar*). En el código se puede diferenciar porque se omite el **var**:

```
var a = 40; // Inicializamos la variable "a" al valor 40.
a = 50; // Ahora, hemos declarado que pasa a contener 50 en lugar de
40.
```

Ámbitos de variables: var

Cuando inicializamos una variable al principio de nuestro programa y le asignamos un valor, ese valor generalmente está disponible a lo largo de todo el programa. Sin embargo, esto puede variar dependiendo de múltiples factores. Se conoce como **ámbito de una variable** a la zona donde esa variable sigue existiendo.

Por ejemplo, si consultamos el valor de una variable antes de inicializarla, no existe:

```
console.log(e); // Muestra "undefined", en este punto la variable "e"
no existe
var e = 40;
console.log(e); // Aquí muestra 40, existe porque ya se ha
inicializado anteriormente
```

En el ejemplo anterior, el ámbito de la variable **e** comienza a partir de su inicialización y "vive" hasta el final del programa. A esto se le llama **ámbito global** y es el ejemplo más sencillo. Más adelante veremos que se va complicando y a veces no resulta tan obvio saber en que ámbito se encuentra.

En el enfoque tradicional de Javascript, es decir, cuando se utiliza la palabra clave **var** para declarar variables, existen dos ámbitos principales: **ámbito global** y **ámbito a nivel de función**.

Observemos el siguiente ejemplo:

```
var a = 1;
console.log(a); // Aquí accedemos a la "a" global, que vale 1

function x() {
  console.log(a); // En esta línea el valor de "a" es undefined
  var a = 5; // Aquí creamos una variable "a" a nivel de función

  console.log(a); // Aquí el valor de "a" es 5 (a nivel de función)
  console.log(window.a); // Aquí el valor de "a" es 1 (ámbito global)
}

x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 1
```

En el ejemplo anterior vemos que el valor de **a** dentro de una función no es el `1` inicial, sino que estamos en otro ámbito diferente donde la variable **a** anterior no existe: un **ámbito a nivel de función**. Mientras estemos dentro de una función, las variables inicializadas en ella estarán en el **ámbito** de la propia función.

OJO: Podemos utilizar el objeto especial **window** para acceder directamente al ámbito global independientemente de donde nos encontremos. Esto ocurre así porque las variables globales se almacenan dentro del objeto **window** (*la pestaña actual del navegador web*).

```
var a = 1;
console.log(a); // Aquí accedemos a la "a" global, que vale 1

function x() {
  console.log(a); // En esta línea el valor de "a" es 1
  a = 5; // Aquí creamos una variable "a" en el ámbito anterior

  console.log(a); // Aquí el valor de "a" es 5 (a nivel de función)
  console.log(window.a); // Aquí el valor de "a" es 5 (ámbito global)
}

x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 5
```

En este ejemplo se omite el **var** dentro de la función, y vemos que en lugar de crear una variable en el ámbito de la función, se modifica el valor de la variable **a** a nivel global. Dependiendo de donde y como accedamos a la **variable a**, obtendremos un valor u otro.

Siempre que sea posible se debería utilizar **let** y **const** (*ver a continuación*), en lugar de **var**. Declarar variables mediante **var** se recomienda en fases de aprendizaje o en el caso de que se quiera mantener compatibilidad con navegadores muy antiguos utilizando ECMAScript 5, sin embargo, hay estrategias mejores a seguir que utilizar **var** en la actualidad.

Ámbitos de variables: let

En las versiones modernas de Javascript (*ES6 o ECMAScript 2015*) o posteriores, se introduce la palabra clave **let** en sustitución de **var**. Con ella, en lugar de utilizar los **ámbitos globales y a nivel de función** (*var*), utilizamos los ámbitos clásicos de programación: **ámbito global y ámbito local**.

La diferencia se puede ver claramente en el uso de un **bucle for** con **var** y con **let**:

```
/** Opción 1: Bucle con let **/

console.log("Antes: ", p);    // Antes: undefined
for (let p = 0; p < 3; p++)
  console.log("- ", p);       // Durante: 0, 1, 2
console.log("Después: ", p);  // Después: undefined

/** Opción 2: Bucle con var **/

console.log("Antes: ", p);    // Antes: undefined
for (var p = 0; p < 3; p++)
  console.log("- ", p);       // Durante: 0, 1, 2
```

```
console.log("Después: ", p);  // Después: 3 (WTF!)
```

Vemos que utilizando **let** la variable **p** sólo existe dentro del bucle, ámbito local, mientras que utilizando **var** la variable **p** sigue existiendo fuera del bucle, ya que debe tener un ámbito global o a nivel de función.

Constantes

De forma tradicional, Javascript no incorporaba constantes. Sin embargo, en ECMAScript 2015 (*ES6*) se añade la palabra clave **const**, que inicializada con un valor concreto, permite crear variables con valores que no pueden ser cambiados.

```
const NAME = "Manz";
console.log(NAME);
```

En el ejemplo anterior vemos un ejemplo de **const**, que funciona de forma parecida a **let**. Una buena práctica es escribir el nombre de la constante en mayúsculas, para identificar rápidamente que se trata de una constante y no una variable, cuando leemos código ajeno.

Realmente, las **constantes** de Javascript son variables inicializadas a un valor específico y que no pueden redeclararse. No confundir con valores inmutables, ya que como veremos posteriormente, los objetos si pueden ser modificados aún siendo constantes.

Objetos básicos

Uno de los aspectos más importantes del lenguaje Javascript es el concepto de **objeto**, puesto que prácticamente todo lo que utilizamos en Javascript, son objetos. Sin embargo, tiene ligeras diferencias con los objetos de otros lenguajes de programación, así que vamos a comenzar con una explicación sencilla y más adelante ampliaremos este tema en profundidad.

¿Qué son los objetos?

En Javascript, existe un tipo de dato llamado **objeto**. No es más que una variable especial que puede contener más variables en su interior. De esta forma, tenemos la posibilidad de organizar múltiples variables de la misma temática dentro de un objeto. Veamos algunos ejemplos:

En muchos lenguajes de programación, para crear un objeto se utiliza la palabra clave `new`. En Javascript también se puede hacer:

```
const objeto = new Object(); // Esto es un objeto «genérico» vacío
```

Sin embargo, siempre que podamos, en Javascript se prefiere utilizar lo que se llaman los **literales**, un método abreviado para crear objetos directamente, sin necesidad de utilizar la palabra `new`.

Declaración de un objeto

Los literales de los objetos en Javascript son las llaves `{}`. Este ejemplo es equivalente al anterior, pero es más corto, rápido y cómodo, por lo que se aconseja declararlos así:

```
const objeto = {}; // Esto es un objeto vacío
```

Pero hasta ahora, solo hemos creado un objeto vacío. Vamos a crear un nuevo objeto, que contenga variables con información en su interior:

```
// Declaración del objeto
const player = {
  name: "Manz",
```

```
  life: 99,
  strength: 10,
};
```

Estas variables dentro de los objetos se suelen denominar **propiedades**. Como se puede ver, un objeto en Javascript nos permite encapsular en su interior información relacionada, para posteriormente poder acceder a ella de forma más sencilla e intuitiva.

Acceso a sus propiedades

Una vez tengamos un objeto, podemos acceder a sus propiedades de dos formas diferentes: a través de la notación con **puntos** o a través de la notación con **corchetes**.

```
// Notación con puntos
console.log(player.name); // Muestra "Manz"
console.log(player.life); // Muestra 99

// Notación con corchetes
console.log(player["name"]); // Muestra "Manz"
console.log(player["life"]); // Muestra 99
```

El programador puede utilizar la notación que más le guste. La más utilizada en Javascript suele ser la **notación con puntos**, mientras que la notación con corchetes se suele conocer en otros lenguajes como «arrays asociativos».

A algunos programadores puede resultarles confuso utilizar objetos con la notación de corchetes, ya que en otros lenguajes de programación los objetos y los arrays asociativos son cosas diferentes, y en Javascript ambos conceptos se mezclan.

Hay ciertos casos en los que sólo se puede utilizar la **notación con corchetes**, como por ejemplo cuando se utilizan espacios en el nombre de la propiedad. Es imposible hacerlo con la notación con puntos.

Añadir propiedades

También podemos añadir **propiedades** al **objeto** después de haberlo creado, aunque la sintaxis cambia ligeramente. Veamos un ejemplo equivalente al anterior:

```
// Declaración del objeto
const player = {};

// Añadimos mediante notación con puntos
player.name = "Manz";
player.life = 99;
player.strength = 10;

// Añadimos mediante notación con corchetes
player["name"] = "Manz";
player["life"] = 99;
player["strength"] = 10;
```

Las **propiedades** del objeto pueden ser utilizadas como variables. De hecho, utilizar los objetos como elementos para organizar múltiples variables suele ser una buena práctica en Javascript.

Tipos de objetos

Hasta ahora, solo hemos visto los objetos «genéricos», en Javascript conocidos como tipo , declarándolos con un `new Object()` o con un literal `{}`, dos formas equivalentes de hacer lo mismo. Al generar una variable de tipo , esa variable «hereda» una serie de métodos (*del objeto `Object` en este caso*).

```
const o = {};
o.toString(); // Devuelve '[object Object]' (Un objeto de tipo Object)
```

En este ejemplo, `toString()` es uno de esos métodos que tienen todas las variables de tipo . Sin embargo, hasta ahora y sin saberlo, cuando creamos una variable de un determinado tipo de dato (*sea primitivo o no*), es también de tipo , ya que todas las variables heredan de este tipo de dato. Por lo tanto, nuestra variable tendrá no sólo los métodos de su tipo de dato, sino también los métodos heredados de :

```
const s = "hola";
s.toString(); // Devuelve 'hola'
```

Más adelante, veremos los métodos que heredan las variables de tipo y comprobaremos que los objetos tienen detrás de si muchos más conceptos que los que hemos visto hasta ahora y que su definición es mucho más amplia.

Variables numéricas (Number)

En Javascript crear variables numéricas es muy sencillo, pero hay que conocer bien como trabajar con ellas y los diferentes métodos de los que dispone.

¿Qué es una variable numérica?

En Javascript, los **números** son uno de los tipos de datos básicos (*tipos primitivos*) que para crearlos, simplemente basta con escribirlos. No obstante, en Javascript todo son objetos, como veremos más adelante, y también se pueden declarar como si fueran un objeto:

Constructor	Descripción
`new Number(n)`	Crea un objeto numérico a partir del número `n` pasado por parámetro.
`n`	Simplemente, el número en cuestión. **Notación preferida**.

Sin embargo, aunque existan varias formas de declararlos, no se suele utilizar la notación `new` con objetos primitivos ya que es bastante más tedioso y complicado que utilizar la notación de literales:

```
// Literales
const n1 = 4;
const n2 = 15.8;

// Objetos
const n1 = new Number(4);
const n2 = new Number(15.8);
```

Cualquier parámetro pasado al `new Number()` que no sea un número, dará como resultado un valor `NaN` (*ver más adelante*).

Constantes numéricas

Existe una serie de constantes definidas en relación a las variables numéricas. La mayoría de ellas establecen límites máximos y mínimos, veamos su significado:

Constante	Valor en Javascript	Descripción
Number.POSITIVE_INFINITY	`Infinity`	Infinito positivo: `+∞`
Number.NEGATIVE_INFINITY	`-Infinity`	Infinito negativo: `-∞`
Number.MAX_VALUE	`1.7976931348623157e+308`	Valor más grande
Number.MIN_VALUE	`5e-324`	Valor más pequeño
Number.MAX_SAFE_INTEGER	`9007199254740991`	Valor seguro más grande
Number.MIN_SAFE_INTEGER	`-9007199254740991`	Valor seguro más

Constante	Valor en Javascript	Descripción
		pequeño
Number.EPSILON	2^{-52}	Número muy pequeño: ε
Number.NaN	`NaN`	Not A Number

La diferencia entre `Number.MAX_VALUE` y `Number.MAX_SAFE_INTEGER` es que, el primero es el **valor máximo** que es posible representar en Javascript. Por otro lado, el segundo es el **valor máximo** para realizar cálculos con seguridad en Javascript.

Los lenguajes de programación están sujetos a la precisión numérica debido a la forma interna en la que guardan valores numéricos. Si necesitamos realizar operaciones con muy alta precisión numérica en Javascript, se recomienda utilizar librerías como decimal.js o bigNumber.js.

NaN (Not A Number)

El acrónimo `NaN` es un valor especial de Javascript que significa **Not A Number** (*No es un número*). Este valor se usa para representar valores imposibles o indeterminados, como por ejemplo, resultados matemáticos de operaciones como:

- `0 / 0` (Indeterminaciones)
- `4 - 'a'` (Valores imposibles)
- `NaN + 4` (Operaciones con `NaN` como operando)

Este valor se utiliza habitualmente para detectar si una operación ha fallado o ha dado un valor no representable. Sin embargo, no podemos compararlo literalmente con `NaN`, sino que tenemos que usar la función `Number.isNaN()`:

```
let num = NaN;

// La siguiente operación, contra toda lógica, es falsa
num == NaN; // false

// Se debe usar Number.isNaN() para comprobar si el valor es NaN
Number.isNaN(num); // true

// Si comprobamos el tipo de dato de NaN, nos dirá que es numérico
typeof num; // number
```

Como se puede ver en la última línea del ejemplo anterior, mencionar que en Javascript, si comprobamos el tipo de dato de `NaN` con **typeof** nos dirá que es un número. Puede parecer ilógico que **Not A Number** sea un número, esto ocurre porque `NaN` está en un contexto numérico.

En otras palabras, dentro de los tipos de datos numéricos, `NaN` es un conjunto de números que no se pueden representar.

Comprobaciones numéricas

En Javascript tenemos varias funciones para conocer la naturaleza de una variable numérica (número finito, número entero, número seguro o si no es representable como un número). Las podemos ver a continuación en la siguiente tabla:

Método	Descripción
Number.isFinite(`n`)	Comprueba si `n` es un número finito.
Number.isInteger(`n`)	Comprueba si `n` es un número entero.
Number.isSafeInteger(`n`)	Comprueba si `n` es un número seguro.
Number.isNaN(`n`)	Comprueba si `n` no es un número.

Ten en cuenta que estas funciones devuelven un booleano (*valor de verdadero o falso*), lo que lo hace ideales para usarlas como condiciones en bucles o condicionales. A continuación veamos dos ejemplos para cada una de estas funciones:

```
// ¿Número finito?
Number.isFinite(42); // true
Number.isFinite(Infinity); // false, es infinito

// ¿Número entero?
Number.isInteger(5); // true
Number.isInteger(4.6); // false, es decimal

// ¿Número seguro?
Number.isSafeInteger(1e15); // true
Number.isSafeInteger(1e16); // false, es un valor no seguro

// ¿No es un número?
Number.isNaN(NaN); // true
Number.isNaN(5); // false, es un número
```

Recuerda no hacer comprobaciones directas con NaN, sino utilizar la función `Number.isNaN()`.

Conversión numérica

En muchos casos tendremos variables de texto que nos interesa convertir a número, para realizar operaciones posteriormente con ellas. Para ello, lo ideal es utilizar las funciones de parseo numérico, `parseInt()` y `parseFloat()`. Veamos cuales son y cómo se pueden utilizar:

Método	Descripción
Number.parseInt(`s`)	Convierte una cadena de texto `s` en un número entero.
Number.parseInt(`s`, `radix`)	Idem al anterior, pero desde una base `radix`.
Number.parseFloat(`s`)	Convierte una cadena de texto `s` en un número decimal.
Number.parseFloat(`s`, `radix`)	Idem al anterior, pero desde una base `radix`.

Para ilustrar esto, veamos un ejemplo con `parseInt()` cuando solo le pasamos un parámetro (*un texto*) que queremos convertir a número:

```
Number.parseInt("42"); // 42
```

```
Number.parseInt("42€"); // 42
Number.parseInt("Núm. 42"); // NaN
Number.parseInt("A"); // NaN
```

Nota que la función `parseInt()` funciona perfectamente para variables de texto que contienen números o que empiezan por números. Esto es muy útil para eliminar unidades de variables de texto. Sin embargo, si la variable de texto comienza por un valor que no es numérico, `parseInt()` devolverá un `NaN`.

Si lo que queremos es quedarnos con el número que aparece más adelante en la variable de texto, habrá que manipular ese texto con alguna de las funciones que veremos en el apartado de variables de texto.

Veamos ahora que ocurre si utilizamos `parseInt()` con dos parámetros, donde el primero es el **texto con el número** y el segundo es la **base numérica** del número:

```
Number.parseInt("11101", 2); // 29 en binario
Number.parseInt("31", 8); // 25 en octal
Number.parseInt("FF", 16); // 255 en hexadecimal
```

Esta modalidad de `parseInt()` se suele utilizar cuando queremos pasar a base decimal un número que se encuentra en otra base (*binaria, octal, hexadecimal*...).

Al igual que con `parseInt()` tenemos otra función llamada `parseFloat()`. Funciona exactamente igual a la primera, sólo que la primera está específicamente diseñada para utilizar con números enteros y la segunda para números decimales. Si utilizamos `parseInt()` con un número decimal, nos quedaremos sólo con la parte entera, mientras que `parseFloat()` la conservará.

Representación numérica

Por último, en el caso de querer cambiar el tipo de representación numérica, podemos utilizar las siguientes funciones para alternar entre **exponencial** y **punto fijo**:

Método	Descripción
.toExponential(`n`)	Convierte el número a notación exponencial con `n` decimales.
.toFixed(`n`)	Convierte el número a notación de punto fijo con `n` decimales.
.toPrecision(`p`)	Utiliza `p` dígitos de precisión en el número.

Observemos el siguiente ejemplo aplicando las funciones anteriores al número decimal `1.5`:

```
(1.5).toExponential(2); // "1.50e+0" en exponencial
(1.5).toFixed(2); // "1.50" en punto fijo
(1.5).toPrecision(1); // "2"
```

Operaciones matemáticas (Math)

Cuando trabajamos con Javascript, es posible realizar gran cantidad de **operaciones matemáticas** de forma nativa, sin necesidad de librerías externas. Para ello, haremos uso del objeto `Math`, un objeto interno de Javascript que tiene incorporadas ciertas constantes y métodos (*funciones*) para trabajar matemáticamente.

Constantes de Math

El objeto `Math` de Javascript incorpora varias constantes que podemos necesitar en algunas operaciones matemáticas. Veamos su significado y valor aproximado:

Constante	Descripción	Valor
`Math.E`	Número de Euler	`2.718281828459045`
`Math.LN2`	Logaritmo natural en base 2	`0.6931471805599453`
`Math.LN10`	Logaritmo decimal	`2.302585092994046`
`Math.LOG2E`	Logaritmo base 2 de E	`1.4426950408889634`
`Math.LOG10E`	Logaritmo base 10 de E	`0.4342944819032518`
`Math.PI`	Número PI o `π`	`3.141592653589793`
`Math.SQRT1_2`	Raíz cuadrada de 1/2	`0.7071067811865476`
`Math.SQRT2`	Raíz cuadrada de 2	`1.4142135623730951`

Además de estas constantes, el objeto `Math` también nos proporciona gran cantidad de métodos o funciones para trabajar con números. Vamos a analizarlos.

Métodos matemáticos

Los siguientes métodos matemáticos están disponibles en Javascript a través del objeto `Math`. Observa que algunos de ellos sólo están disponibles en **ECMAScript 6**:

Método	Descripción	Ejemplo
`Math.abs(x)`	Devuelve el valor absoluto de `x`.	`|x|`
`Math.sign(x)`	Devuelve el signo del número: `1` positivo, `-1` negativo	
`Math.exp(x)`	Exponenciación. Devuelve el número `e` elevado a `x`.	e^x
`Math.expm1(x)`	Equivalente a `Math.exp(x) - 1`.	e^x-1
`Math.max(a, b, c...)`	Devuelve el número más grande de los indicados por parámetro.	
`Math.min(a, b, c...)`	Devuelve el número más pequeño de los indicados por parámetro.	
`Math.pow(base, exp)`	Potenciación. Devuelve el número `base` elevado a `exp`.	$base^{exp}$

Método	Descripción	Ejemplo
`Math.sqrt(x)`	Devuelve la raíz cuadrada de `x`.	$\sqrt{x}$
`Math.cbrt(x)`	Devuelve la raíz cúbica de `x`.	$\sqrt[3]{x}$
`Math.imul(a, b)`	Equivalente a `a * b`, pero a nivel de bits.	
`Math.clz32(x)`	Devuelve el número de ceros a la izquierda de `x` en binario (32 bits).	

Veamos algunos ejemplos aplicados a las mencionadas funciones anteriormente:

```
Math.abs(-5); // 5
Math.sign(-5); // -1
Math.exp(1); // e, o sea, 2.718281828459045
Math.expm1(1); // 1.718281828459045
Math.max(1, 40, 5, 15); // 40
Math.min(5, 10, -2, 0); // -2
Math.pow(2, 10); // 1024
Math.sqrt(2); // 1.4142135623730951
Math.cbrt(2); // 1.2599210498948732
Math.imul(0xffffffff, 7); // -7

// Ejemplo de clz32 (count leading zeros)
const x = 1;
"0".repeat(Math.clz32(x)) + x.toString(2);
// Devuelve "00000000000000000000000000000001"
```

Existe uno más, `Math.random()` que merece una explicación más detallada, por lo que lo explicamos en el apartado siguiente.

Método Math.random()

Uno de los métodos más útiles e interesantes del objeto `Math` es `Math.random()`.

Método	Descripción	Ejemplo
`Math.random()`	Devuelve un número al azar entre `0` y `1` con 16 decimales.	

Este método nos da un número al azar entre los valores `0` y `1`, con 16 decimales. Normalmente, cuando queremos trabajar con números aleatorios, lo que buscamos es obtener un número entero al azar entre `a` y `b`. Para ello, se suele hacer lo siguiente:

```
// Obtenemos un número al azar entre [0, 1) con 16 decimales
let x = Math.random();

// Multiplicamos dicho número por el valor máximo que buscamos (5)
x = x * 5;

// Redondeamos inferiormente, quedándonos sólo con la parte entera
x = Math.floor(x);
```

Este ejemplo nos dará en `x` un valor al azar entre `0` y `5` (*5 no incluido*). Lo hemos realizado por pasos para entenderlo mejor, pero podemos realizarlo directamente como se ve en el siguiente ejemplo:

```
// Número al azar entre 0 y 5 (no incluido)
const x = Math.floor(Math.random() * 5);

// Equivalente al anterior
const x = ~~(Math.random() * 5);
```

Como se puede ver en el segundo ejemplo anterior, utilizamos el operador a nivel de bits `~~` (*doble negación*) como reemplazo rápido de `Math.floor()`, una función que realiza un redondeo inferior, y que veremos al final de este tema.

Si lo deseas, puedes utilizar librerías específicas para generar números aleatorios como random.js o chance.js, esta última permitiendo incluso generar otros tipos de datos aleatorios como textos, GUIDs o colores hexadecimales.

Métodos de logaritmos

Javascript incorpora varios métodos en el objeto `Math` para trabajar con logaritmos. Desde **logaritmos neperianos** hasta **logaritmos binarios** a través de las siguientes funciones:

Método	Descripción y Ejemplo
`Math.log(x)`	Devuelve el logaritmo natural en base `e` de `x`. **Ej:** $\log_e x$ o `ln x`
`Math.log10(x)`	Devuelve el logaritmo decimal (en base 10) de `x`. **Ej:** $\log_{10} x$ ó `log x`
`Math.log2(x)`	Devuelve el logaritmo binario (en base 2) de `x`. **Ej:** $\log_2 x$
`Math.log1p(x)`	Devuelve el logaritmo natural de `(1+x)`. **Ej:** $\log_e (1+x)$ o `ln (1+x)`

A continuación, unos ejemplos de estas funciones aplicadas:

```
Math.log(2); // 0.6931471805599453
Math.log10(2); // 0.3010299956639812
Math.log2(2); // 1
Math.log1p(2); // 1.0986122886681096
```

Métodos de redondeo

Como hemos visto anteriormente, es muy común necesitar métodos para **redondear números** y reducir el número de decimales o aproximar a una cifra concreta. Para ello, de forma nativa, Javascript proporciona los siguientes métodos de redondeo:

Método	Descripción
`Math.round(x)`	Devuelve el redondeo de `x` (*el entero más cercano*)
`Math.ceil(x)`	Devuelve el redondeo superior de `x`. (*el entero más alto*)
`Math.floor(x)`	Devuelve el redondeo inferior de `x`. (*el entero más bajo*)
`Math.fround(x)`	Devuelve el redondeo de `x` (*flotante con precisión simple*)
`Math.trunc(x)`	Trunca el número `x` (*devuelve sólo la parte entera*)

Veamos las diferencias de utilizar los diferentes **métodos** anteriores para redondear un número decimal y los resultados obtenidos:

```
// Redondeo natural, el más cercano
Math.round(3.75); // 4
Math.round(3.25); // 3

// Redondeo superior (el más alto)
Math.ceil(3.75); // 4
Math.ceil(3.25); // 4

// Redondeo inferior (el más bajo)
Math.floor(3.75); // 3
Math.floor(3.25); // 3

// Redondeo con precisión
Math.round(3.123456789); // 3
Math.fround(3.123456789); // 3.1234567165374756

// Truncado (sólo parte entera)
Math.trunc(3.75); // 3
Math.round(-3.75); // -4
Math.trunc(-3.75); // -3
```

Métodos trigonométricos

Por último, y no por ello menos importante, el objeto `Math` nos proporciona de forma nativa una serie de métodos trigonométricos, que nos permiten hacer cálculos con operaciones como **seno**, **coseno**, **tangente** y relacionados:

Método	Descripción
`Math.sin(x)`	Seno de `x`
`Math.asin(x)`	Arcoseno de `x`
`Math.sinh(x)`	Seno hiperbólico de `x`
`Math.asinh(x)`	Arcoseno hiperbólico de `x`
`Math.cos(x)`	Coseno de `x`
`Math.acos(x)`	Arcocoseno de `x`
`Math.cosh(x)`	Coseno hiperbólico de `x`
`Math.acosh(x)`	Arcocoseno hiperbólico de `x`
`Math.tan(x)`	Tangente de `x`
`Math.atan(x)`	Arcotangente de `x`
`Math.tanh(x)`	Tangente hiperbólica de `x`
`Math.atanh(x)`	Arcotangente hiperbólica de `x`
`Math.atan2(x, y)`	Arcotangente del conciente de `x/y`
`Math.hypot(a, b..)`	Devuelve la raíz cuadrada de $a^2 + b^2 + ...$

Otras librerías matemáticas

Si de forma nativa no encuentras una forma sencilla de resolver el problema matemático que tienes entre manos, no olvides que existen una serie de **librerías de terceros** que pueden hacernos la vida más fácil a la hora de trabajar con otros valores matemáticos.

A continuación, detallamos algunas de ellas:

Librería	Descripción	GitHub
Math.js	Librería matemática de propósito general.	josdejong/mathjs
Fraction.js	Librería matemática para trabajar con fracciones.	infusion/Fraction.js
Polynomial.js	Librería matemática para trabajar con polinomios.	infusion/Polynomial.js
Complex.js	Librería matemática para trabajar con números complejos.	infusion/Complex.js
Angles.js	Librería matemática para trabajar con ángulos.	infusion/Angles.js
BitSet.js	Librería matemática para trabajar con vectores de bits.	infusion/BitSet.js

Habrás comprobado que, al contrario que muchos otros objetos de Javascript, en estas ocasiones hemos indicado explícitamente el objeto, por ejemplo `Math.round(numero)`, en lugar de hacerlo sobre la variable: `numero.round()`. Esto ocurre porque **Math** es un objeto con métodos y constantes **estáticas**, algo que veremos en profundidad en futuros temas.

Variables de texto (String)

En Javascript y en el mundo del desarrollo web, una de las tareas que más solemos hacer es manejar cadenas de texto y realizando tareas con ellas. Por lo tanto, tenemos que familiarizarnos con el tipo de dato .

¿Qué es un string?

En programación, cuando hablamos de una varible que posee información de texto, decimos que su tipo de dato es . En Javascript, es muy sencillo crear una variable de texto, hay dos formas de hacerlo:

Constructor	Descripción
`new String(s)`	Crea un objeto de texto a partir del texto `s` pasado por parámetro.
`'s'`	Simplemente, el texto entre comillas. **Notación preferida**.

Los son tipos de datos primitivos, y como tal, es más sencillo utilizar los literales que la notación con `new`. Para englobar los textos, se pueden utilizar **comillas simples** `'`, **comillas dobles** `"` o **backticks** `` ` `` (*ver más adelante*).

Aunque es posible utilizar comillas simples o comillas dobles en los , se recomienda decantarse por uno de los dos estilos y no mezclarlos. Muchas empresas o equipos de desarrollo tienen guías de estilos para delimitar cuál utilizar.

A continuación, un ejemplo de declaración de variables de texto en Javascript:

```
// Literales
const texto1 = "¡Hola a todos!";
const texto2 = "Otro mensaje de texto";

// Objeto
const texto1 = new String("¡Hola a todos!");
const texto2 = new String("Otro mensaje de texto");
```

A diferencia de otros lenguajes de programación, que separan el tipo de dato (*cadena de texto*) del tipo de dato **char** (*un solo carácter*), Javascript los mantiene a ambos dentro del tipo de dato , por lo que una variable, aunque sólo contenga un carácter, sigue siendo un .

Propiedades

Al crear una variable con contenido de texto, o sea un , automáticamente, esa variable pasa a tener a su disposición todas las propiedades y métodos disponibles para este tipo

de dato, por lo que podemos utilizarlos en nuestro código como iremos viendo a continuación.

Propiedad	Descripción
`.length`	Devuelve el número de carácteres de la variable de tipo string en cuestión.

En el caso de los strings, solo tenemos una propiedad, `.length`, que devuelve el tamaño de la variable de texto en cuestión. Nótese en los siguientes ejemplos que se han utilizado directamente, sin necesidad de guardarlos en una variable antes:

```
"Hola".length; // 4
"Adiós".length; // 5
"".length; // 0
"¡Yeah!".length; // 6
```

Pero las características más interesantes de los strings se encuentran en los métodos de dicho tipo de dato. Ten en cuenta que, en las variables de texto, los métodos se ejecutan sobre el propio texto del , devolviendo información modificada a partir de este. Vamos a dar un repaso a los métodos que existen.

Métodos de posiciones

En primer lugar existen varios métodos que permiten darnos información sobre la **posición** o **ubicación** que ocupa un determinado carácter o texto. Esta posición también suele denominarse **índice**. Veamos detalladamente dicho métodos:

Método	Descripción	Oper.
`.charAt(pos)`	Devuelve el carácter en la posición `pos` de la variable.	`[]`
`.concat(str1, str2...)`	Devuelve el texto de la variable unido a `str1`, a `str2`...	`+`
`.indexOf(str)`	Devuelve la primera posición del texto `str`.	
`.indexOf(str, from)`	Idem al anterior, partiendo desde la posición `from`.	
`.lastIndexOf(str, from)`	Idem al anterior, pero devuelve la última posición.	

El método `.charAt(pos)` nos permite comprobar que carácter se encuentra en la posición `pos` del texto. Este método devolverá un con dicho carácter. En caso de pasarle una posición que no existe o imposible (*negativa*), simplemente nos devolverá un **vacío**. El valor por defecto de `pos` es `0`.

No obstante, es preferible utilizar el operador `[]` para obtener el carácter que ocupa una posición, ya que es más corto y rápido de utilizar y mucho más claro. La diferencia respecto a `charAt()` es que el operador `[]` devuelve si en esa posición no existe ningún carácter.

```
"Manz".charAt(0); // 'M'
"Manz".charAt(1); // 'a'
"Manz".charAt(10); // ''
"Manz"[0]; // 'M'
```

```
"Manz"[1]; // 'a'
"Manz"[10]; // undefined
```

El método `.concat(str1, str2...)` permite concatenar (*unir*) los textos pasados por parámetros al de la propia variable. Al igual que el método anterior, es preferible utilizar el operador `+`, ya que es mucho más rápido y legible. Mucho cuidado con utilizar el operador `+`, ya que depende de los tipos de datos con los que se usen puede producir un efecto diferente. El operador `+` usado con realiza concatenaciones, mientras que usado con realiza sumas.

```
"Manz".concat("i", "to"); // 'Manzito'
"Manz" + "i" + "to"; // 'Manzito'
"Manz" + 4 + 5; // 'Manz45'
10 + 5 + 4 + 5; // 24
```

Por último, nos queda el método `indexOf(str, from)`, que es la función opuesta a `charAt()`. La función `indexOf(str)` buscará el subtexto `str` en nuestra variable y nos devolverá un con la **posición** de la primera aparición de dicho subtexto. En caso de no encontrarlo, devolverá `-1`. El parámetro `from` es opcional, y es la posición en la que empezará a buscar, que por defecto (*si no se suministra*) es `0`.

```
"LenguajeJS, página de Javascript".indexOf("n"); // 2
"LenguajeJS, página de Javascript".indexOf("n", 3); // 16
"LenguajeJS, página de Javascript".indexOf("n", 17); // -1
"LenguajeJS, página de Javascript".lastIndexOf("n"); // 16
"LenguajeJS, página de Javascript".lastIndexOf("n", 3); // 2
```

El método `lastIndexOf(str, from)` funciona exactamente igual que el anterior, sólo que realiza la búsqueda de la **última aparición** en lugar de la primera aparición.

Métodos para búsquedas

Los siguientes métodos se utilizan para realizar búsquedas o comprobaciones de subtextos en el texto de un :

Método	Descripción
`.startsWith(s, from)`	Comprueba si el texto comienza por `s` desde la posición `from`.
`.endsWith(s, to)`	Comprueba si el texto hasta la posición `to`, termina por `s`.
`.includes(s, from)`	Comprueba si el texto contiene el subtexto `s` desde la posición `from`.
`.search(regex)`	Busca si hay un patrón que encaje con `regex` y devuelve la posición.
`.match(regex)`	Idem a la anterior, pero devuelve las coincidencias encontradas.

Por ejemplo, el método `startsWith(str, from)` devolverá `true` si la variable comienza por el texto proporcionado en `str`. Si además se indica el parámetro opcional `from`, empezará en la posición `from` del . De la misma forma, el método `endsWith()` comprueba cuando un acaba en `str`, y el método `includes()` comprueba si el subtexto dado está incluído en el .

Algunos ejemplos:

```
"Manz".startsWith("M"); // true ('Manz' empieza por 'M')
"Manz".startsWith("a", 1); // true ('anz' empieza por 'a')
"Manz".endsWith("o"); // false ('Manz' no acaba en 'o')
"Manz".endsWith("n", 3); // true ('Man' acaba en 'n')
"Manz".includes("an"); // true ('Manz' incluye 'an')
"Manz".includes("M", 1); // false ('anz' no incluye 'M')
```

Por otro lado, los métodos `search()` y `match()` realizan búsquedas más potentes y flexibles con . La diferencia de cada una es que, mientras el método `search()` devuelve la posición, `matches()` devuelve un con las coincidencias.

```
// La expresión regular /o/g busca globalmente las "o" en el texto

"Hola a todos".search(/o/g); // 1, porque la primera "o" está en la
posición 1
"Hola a todos".match(/o/g); // ['o', 'o', 'o'], las 3 "o" que
encuentra
```

Las **expresiones regulares** permiten realizar cosas mucho más avanzadas que las que se muestran en este ejemplo, por lo tanto, las abordaremos en temas posteriores.

Métodos para transformar

En Javascript podemos utilizar algunos métodos para modificar un realizando alguna operación de transformación. En esta tabla tenemos dichos métodos:

Método	Descripción
`.repeat(n)`	Devuelve el texto de la variable repetido `n` veces.
`.toLowerCase()`	Devuelve el texto de la variable en minúsculas.
`.toUpperCase()`	Devuelve el texto de la variable en mayúsculas.
`.trim()`	Devuelve el texto sin espacios a la izquierda y derecha.
`.trimStart()`	Devuelve el texto sin espacios a la izquierda.
`.trimEnd()`	Devuelve el texto sin espacios a la derecha.
`.replace(str|regex, newstr)`	Reemplaza la primera aparición del texto `str` por `newstr`.
`.replaceAll(str|regex, newstr)`	Reemplaza todas las apariciones del texto `str` por `newstr`.
`.replace(str|regex, func)`	Idem a `.replace()`, pero reemplazando por la devolución de `func`.
`.substr(ini, len)`	Devuelve el subtexto desde la posición `ini` hasta `ini+len`.
`.substring(ini, end)`	Devuelve el subtexto desde la posición `ini` hasta `end`.
`.slice(ini, end)`	Idem a `.substr()` con leves diferencias.
`.split(sep|regex, limit)`	Separa el texto usando `sep` como separador, en `limit` fragmentos.

Método	Descripción
`.padStart(len, str)`	Rellena el principio de la cadena con `str` hasta llegar al tamaño `len`.
`.padEnd(len, str)`	Rellena el final de la cadena con `str` hasta llegar al tamaño `len`.

El método `repeat(n)` devuelve como el texto repetido `n` veces. Por otro lado, los métodos `toLowerCase()` y `toUpperCase()` devuelven el texto convertido todo a minúsculas o todo a mayúsculas respectivamente:

```
"Na".repeat(5); // 'NaNaNaNaNa'
"MANZ".toLowerCase(); // 'manz'
"manz".toUpperCase(); // 'MANZ'
"    Hola  ".trim(); // 'Hola'
```

Por último, el método `trim()`, informalmente traducido como «afeitar» se encarga de devolver el texto eliminando los espacios sobrantes que hay a la izquierda o a la derecha del texto (*y sólo esos, nunca los que hay entre palabras*). De la misma forma, `trimStart()` y `trimEnd()` realizan la misma tarea sólo a la izquierda y sólo a la derecha respectivamente.

Reemplazar textos

Uno de los métodos más interesantes de transformación de es el `replace(str, newstr)`. Su funcionalidad más básica, como se ve en el primer ejemplo, se trata de devolver el texto en cuestión, reemplazando el texto `str` por `newstr` (*¡y solo la primera aparición!*):

```
"Amigo".replace("A", "Ene"); // 'Enemigo'
"Dispara".replace("a", "i"); // 'Dispira' (sólo reemplaza la primera aparición)
"Dispara".replace(/a/g, "i"); // 'Dispiri' (reemplaza todas las ocurrencias)
```

Si lo que nos interesa es reemplazar todas las apariciones, tendremos que hacer uso de las **expresiones regulares**, que veremos en temas posteriores a este. A grandes rasgos, en el tercer ejemplo anterior, en lugar de indicar el string `'a'` indicamos la expresión regular `/a/g` que buscará todas las apariciones de `a` de forma global (*todas las ocurrencias*).

Desde **ECMAScript** {.es2021} es posible utilizar `replaceAll()` para reemplazar **todas** las ocurrencias de un texto o de una expresión regular. Funciona exactamente igual que `replace()`, sólo que reemplaza todas las ocurrencias en vez de solamente la primera.

Además, el método `replace()` nos permite indicar, como segundo parámetro una en lugar de un , permitiendo utilizar dicha función para realizar un proceso más complejo al reemplazar, en lugar de simplemente reemplazar por un . Sin embargo, para aprender a utilizar esta funcionalidad, antes tendremos que aprender los **callbacks**, que veremos también más adelante.

Extraer subtextos

Otras de las operaciones fundamentales de los es la posibilidad de extraer pequeños fragmentos de texto de textos más grandes. Para ello tenemos dos aproximaciones para realizarlo: con el método `substr()` o con el método `substring()`.

En el primer caso, el método `substr(ini, len)` nos solicita dos parámetros, `ini`, que es la posición inicial del subtexto, y `len`, que es el tamaño o longitud que tendrá el texto. De esta forma, `substr(2, 4)` extrae el fragmento de texto desde la posición `2` y desde esa posición `4` posiciones más. En el caso de omitirse el parámetro `len`, se devuelve el subtexto hasta el final del texto original:

```
"Submarino".substr(3); // 'marino' (desde el 3 en adelante)
"Submarino".substr(3, 1); // 'm'      (desde el 3, hasta el 3+1)
"Submarino".substring(3); // 'marino' (desde el 3 en adelante)
"Submarino".substring(3, 6); // 'mar'    (desde el 3, hasta el 6)
```

Por otro lado, el método `substring(ini, end)` extrae el fragmento de texto desde la posición `ini` hasta la posición `end`. De igual forma al anterior, si se omite el parámetro `end`, el subtexto abarcará hasta el final del texto original.

Crear Arrays a partir de textos

Otro método muy útil es `split(sep)`, un método que permite **dividir** un por el substring `sep` como separador, devolviendo un array con cada una de las partes divididas. Es muy útil para **crear arrays**, o dividir en diferentes secciones textos que tienen **separadores** repetidos como podrían ser comas, puntos o pipes:

```
"1.2.3.4.5".split("."); // ['1', '2', '3', '4', '5'] (5 elementos)
"Hola a todos".split(" "); // ['Hola', 'a', 'todos'] (3 elementos)
"Código".split(""); // ['C', 'ó', 'd', 'i', 'g', 'o'] (6 elementos)
```

En el último ejemplo, el separador es una **cadena vacía**, es decir, «ningún carácter». Si le indicamos a `split()` que separe por «ningún carácter», lo que hace es hacer una división en su unidad mínima, carácter por carácter.

En el tema de los arrays veremos un método llamado `join()` que es justo el opuesto de `split()`. Si **split** separa un string en varios y los mete en un array, **join** une varios elementos de un array añadiéndole un separador y lo convierte en string.

Relleno de cadenas

Otra transformación interesante con los es la resultante de utilizar métodos como `padStart(len, str)` o `padEnd(len, str)`. Ambos métodos toman dos parámetros: `len` la longitud deseada del resultante y `str` el carácter a utilizar como relleno.

El objetivo de ambas funciones es devolver un nuevo con la información original existente, pero ampliando su tamaño a `len` y rellenando el resto con `str`, al principio si se usa `padStart()` o al final si se usa `padEnd()`:

```
"5".padStart(6, "0"); // '000005'
"A".padEnd(5, "·"); // 'A····'
```

Estos métodos resultan especialmente interesantes para formatear horas, como por ejemplo en el caso que queremos que las cifras menores a 10 aparezcan en formato `00` en lugar de `0`.

Métodos Unicode

Unicode es el nombre por el que se conoce al sistema moderno de codificación de carácteres que se usa en informática. A grandes rasgos, cada carácter como podría ser la `A`, la `B` o cualquier otro, tiene su representación **Unicode**, que se basa en un código o **code point**.

Por ejemplo, el carácter **A** corresponde al código Unicode `U+0041`. Este `0041` realmente está en hexadecimal, por lo que `0x0041` en decimal sería igual a `65`. Existen muchísimos códigos, ya que cualquier carácter existente, tiene su propio código Unicode. En Javascript, tenemos dos métodos interesantes relacionado con este tema:

Método	Descripción
`String.fromCharCode(num)`	Devuelve el carácter del valor **unicode** indicado en `num`.
`.charCodeAt(pos)`	Devuelve el valor **unicode** del carácter de la posición `pos` del texto.

El primero de ellos es un método estático, por lo que hay que escribir directamente `String.fromCharCode()` y no utilizarlo desde una variable. Para usar este método, le pasamos un `num` por parámetro, que indicará el número o código Unicode al que queremos hacer referencia, y el método nos devolverá un con el carácter Unicode en cuestión:

```
String.fromCharCode(65); // 'A' (65 es el código U+0041 en decimal)
String.fromCharCode(0x0041); // 'A' (0x0041 es el código U+0041 en
hexadecimal)
"A".charCodeAt(0); // 65
"A".charCodeAt(0).toString(16); // 41
```

Por otro lado, el método `charCodeAt()` es la operación inversa a `String.fromCharCode()` con algún extra. A `charCodeAt(pos)` le pasamos una posición con `pos` por parámetro. Esto buscará el carácter de la posición `pos` del y nos devolverá su código Unicode (*por defecto, en decimal*). Si queremos pasarlo a otra base numérica, podemos hacer uso del método `toString(base)` indicando `16` como `base`.

Observa a continuación que, los famosos **emojis** (*por ejemplo*), son realmente una combinación de 2 códigos Unicode:

```
// El valor unicode del emoji 🐓 es (55357, 56358)
emoji = "🐓";
codigos = [];

for (let i = 0; i < emoji.length; i++) {
  codigos.push(emoji.charCodeAt(i));
}
```

```
String.fromCharCode(...codigos); // '🐓' (Usamos desestructuración, ver más adelante)
```

Observa que modificando el último código Unicode, podemos obtener diferentes **emojis**:

```
String.fromCharCode(55357, 56358); // '🐦'
String.fromCharCode(55357, 56359); // '🐧'
String.fromCharCode(55357, 56360); // '🐨'
("\u0041"); // 'A'
("\ud83d\udc28"); // '🐨'
```

Una forma rápida de escribir **carácteres Unicode** es utilizando la secuencia de escape `\u` seguida del código Unicode en hexadecimal del caracter en cuestión, como se ve en los dos últimos ejemplos anteriores.

Interpolación de variables

En **ECMAScript** se introduce una interesante mejora en la manipulación general de , sobre todo respecto a la legibilidad de código.

Hasta ahora, si queríamos concatenar el valor de algunas variables con textos predefinidos por nosotros, teníamos que hacer algo parecido a esto:

```
const sujeto = "frase";
const adjetivo = "concatenada";
"Una " + sujeto + " bien " + adjetivo; // 'Una frase bien concatenada'
```

A medida que añadimos más variables, el código se hace bastante menos claro y más complejo de leer, especialmente si tenemos que añadir arrays, introducir comillas simples que habría que escapar con `\'` o combinar comillas simples con dobles, etc...

Para evitarlo, se introducen las **backticks** (*comillas hacia atrás*), que nos permiten **interpolar** el valor de las variables sin tener que cerrar, concatenar y abrir la cadena de texto continuamente:

```
const sujeto = "frase";
const adjetivo = "concatenada";
`Una ${sujeto} mejor ${adjetivo}`; // 'Una frase mejor concatenada'
```

Esto es una funcionalidad muy simple, pero que mejora sustancialmente la calidad de código generado. Eso sí, recuerda que se introduce en **ECMAScript 6**, con todo lo que ello conlleva

Fechas nativas (Date)

En muchas ocasiones necesitaremos guardar o trabajar con fechas en nuestros programas. Una fecha tiene datos mixtos: día, mes y año, pero también puede ser más precisa y tener hora, minutos y/o segundos. Además, la hora puede estar en varios formatos.

Toda esta información no se podría guardar en una sola variable numérica, y tampoco es apropiado guardar en formato de texto, porque luego no podríamos hacer cálculos de fechas con ella. Para ello, tenemos un objeto llamado **Date** que nos vendrá perfecto para estos casos.

¿Qué es el tipo de dato Date?

Javascript nos provee de un tipo de dato llamado **Date**, con el que podemos trabajar facilmente con fechas de forma nativa y práctica. Sin embargo, trabajar con fechas no es fácil y la primera vez que tenemos que hacerlo es muy fácil equivocarse si no tenemos claros algunos conceptos.

Lo primero es ver los constructores para saber como podemos construir una variable de tipo **Date**:

Constructor	Descripción
`new Date()`	Obtiene la fecha del momento actual.
`new Date(str)`	Convierte el texto con formato `YYYY/MM/DD HH:MM:SS` a fecha.
`new Date(num)`	Convierte el número `num`, en formato **Tiempo UNIX**, a fecha UTC.
`new Date(y, m, d, h, min, s, ms)`	Crea una fecha UTC a partir de componentes numéricos*.

Podemos utilizar estas cuatro formas para crear fechas en Javascript. Observa que en algunos casos se menciona **fecha UTC**. De momento, vamos a obviar esta parte y más adelante profundizaremos en ella. Veamos algunos ejemplos para crear fechas con estos 4 constructores:

```
// Obtenemos la fecha actual y la guardamos en la variable f
const f = new Date();

// Obtenemos la fecha 30 de Enero de 2018, a las 23h 30m 14seg
const f = new Date("2018/01/30 23:30:14");

// Obtenemos la fecha del juicio final a partir de un timestamp o
Tiempo UNIX
new Date(872817240000);
```

```
// Creamos una fecha pasando cada uno de sus componentes numéricos*
new Date(y, m, d, h, min, s, ms);
```

OJO: Mucho cuidado con los **«componentes numéricos»** mencionados en el último ejemplo. Si utilizamos el formato `new Date(y, m, d, h, min, s, ms)`, hay que saber que, como mínimo, los parámetros `y` (*año*) y `m` (*mes*) son **obligatorios**, el resto son parámetros opcionales.

Además, si utilizamos este esquema, hay que tener en cuenta que hay ciertas características especiales:

- El parámetro `m` (*mes*) se proporciona con valores no reales que se recalcularán. Es decir, si indicamos un `1` nos referimos a **febrero** y no a **enero**. Si queremos referirnos a **enero** tenemos que indicar un `0`.
- Si indicamos **valores negativos** contabilizamos hacia atrás. Por ejemplo, con `2018, -1` estaríamos indicando **diciembre de 2017**. De la misma forma, `2018, 12` haría referencia a **enero de 2019**. Lo mismo ocurre con otros parámetros; `2018, 0, 32` haría referencia al 1 de febrero de 2018.

Tiempo UNIX

El **Tiempo UNIX** (*o UNIX timestamp*) es un formato numérico utilizado para calcular una fecha en UNIX. Es una forma poco práctica y legible para humanos, pero muy eficiente en términos informáticos. Se trata de un número que representa la cantidad de **segundos** transcurridos desde la fecha `1/1/1970, a las 00:00:00`.

Así pues, siendo números, resulta muy fácil trabajar y operar con ellos. Una fecha `A` y una fecha posterior `B`, si hacemos `B - A` nos devuelve el número de segundos transcurridos entre ambas fechas, con lo que se podría sacar la diferencia de tiempo.

No obstante, el **Tiempo UNIX** sirve para trabajar con fechas a bajo nivel. Si lo deseas, al final del tema encontrarás una tabla con librerías más cómodas y prácticas para trabajar con fechas en Javascript.

Existen dos métodos que se pueden utilizar para crear fechas, al margen de los constructores anteriores, sólo que estos devuelven directamente el `Tiempo UNIX` de la fecha especificada:

Método	Descripción
`Date.now()`	Devuelve el `Tiempo UNIX` de la fecha actual. Equivalente a `+new Date()`.
`Date.parse(str)`	Convierte un de fecha a `Tiempo UNIX`. Equivalente a `+new Date(str)`.

Veamos algunos ejemplos aplicados para entenderlos:

```
// Estas tres operaciones son equivalentes
const f = Date.now();
const f = +new Date();
```

```
const f = new Date().getTime();

// Estas tres operaciones son equivalentes
const f = Date.parse("2018/10/30");
const f = +new Date("2018/10/30");
const f = new Date("2018/10/30").getTime();
```

- En el primer caso, utilizamos directamente los métodos estáticos `now()` y `parse()` para obtener el número con el **tiempo Unix**.
- En el segundo caso, creamos un objeto Date con `new Date()`, lo que devuelve una fecha. Sin embargo, en Javascript, podemos preceder a esa fecha con el símbolo `+`, obligándolo a evaluarlo de forma numérica, lo que hace que obtenga el **tiempo Unix** a partir de la fecha.
- En el tercer caso, escribimos en una línea dos acciones: crear la variable de fecha con `new Date()` y posteriormente, sobre esa fecha, hacemos un `getTime()`, que nos devuelve el **tiempo Unix** de un Date, como veremos más adelante.

Getters: Obtener fechas

Una vez hemos creado una fecha y tenemos el objeto , podemos trabajar muy facilmente con estas variables a través de sus sencillos métodos. Los siguientes son una lista de **getters**, funciones para obtener información, sobre la fecha almacenada:

Método	Descripción
`.getDay()`	Devuelve el día de la semana: **OJO:** `0` Domingo, `6` Sábado.
`.getFullYear()`	Devuelve el año con 4 cifras.
`.getMonth()`	Devuelve la representación interna del mes. **OJO:** `0` Enero - `11` Diciembre.
`.getDate()`	Devuelve el día del mes.
`.getHours()`	Devuelve la hora. **OJO:** Formato militar; `23` en lugar de `11`.
`.getMinutes()`	Devuelve los minutos.
`.getSeconds()`	Devuelve los segundos.
`.getMilliseconds()`	Devuelve los milisegundos.
`.getTime()`	Devuelve el UNIX Timestamp: segundos transcurridos desde `1/1/1970`.
`.getTimezoneOffset()`	Diferencia horaria (*en min*) de la hora local respecto a UTC (ver más adelante).

Algunos ejemplos del uso de estos métodos serían los siguientes:

```
const f = new Date("2018/01/30 15:30:10.999");

f.getDay(); // 2 (Martes)
f.getDate(); // 30
f.getMonth(); // 0 (Enero)
f.getFullYear(); // 2018
f.getHours(); // 15
f.getMinutes(); // 30
f.getSeconds(); // 10
```

```
f.getMilliseconds(); // 999
f.getTimezoneOffset(); // 0
f.getTime(); // 1517326210999 (Tiempo Unix)
```

Observa que Javascript no tiene forma de devolver, por ejemplo, el día de la semana o el mes en formato de texto. Sin embargo, ello se puede hacer de forma muy **sencilla** utilizando un array:

```
const MESES = [
  "Enero",
  "Febrero",
  "Marzo",
  "Abril",
  "Mayo",
  "Junio",
  "Julio",
  "Agosto",
  "Septiembre",
  "Octubre",
  "Noviembre",
  "Diciembre",
];
const f = new Date();

MESES[f.getMonth()]; // Devuelve el mes actual en formato de texto
```

Curiosidad: El método `getYear()` se utilizaba para devolver la fecha con formato de 2 cifras. Se dejó de usar debido a la llegada del efecto 2000, reemplazándose por el método `getFullYear()` que usa el formato de 4 cifras.

Setters: Cambiar fechas

De la misma forma que en el apartado anterior podemos obtener fechas específicas, con los **setters** podemos modificarlas o alterarlas.

Método	Descripción
`.setFullYear(year)`	Altera el año de la fecha, cambiándolo por `year`. Formato de 4 dígitos.
`.setMonth(month)`	Altera el mes de la fecha, cambiándolo por `month`. **Ojo**: `0-11` (Ene-Dic).
`.setDate(day)`	Altera el día de la fecha, cambiándolo por `day`.
`.setHour(hour)`	Altera la hora de la fecha, cambiándola por `hour`.
`.setMinutes(min)`	Altera los minutos de la fecha, cambiándolos por `min`.
`.setSeconds(sec)`	Altera los segundos de la fecha, cambiándolos por `sec`.
`.setMilliseconds(ms)`	Altera los milisegundos de la fecha, cambiándolos por `ms`.

Nota: Además de cambiar la fecha del objeto en cuestión, estos métodos devuelven un en **tiempo Unix**, con la fecha modificada.

De la misma forma que mencionamos anteriormente, hay que tener en cuenta que los valores pasados a estos métodos pueden recalcular fechas: `setMonth(0)` implica `Enero`,

`setDate(0)` implica último día del mes anterior, `setDate(-1)` implica penúltimo día del mes anterior, etc...

```
const f = new Date("2018/01/30 15:30:10.999");

f.setDate(15); // Cambia a 15/01/2018 15:30:10.999 (Devuelve
1516030210999)
f.setMonth(1); // Cambia a 15/02/2018 15:30:10.999 (Devuelve
1518708610999)
f.setFullYear(2020); // Cambia a 15/02/2020 15:30:10.999 (Devuelve
1581780610999)
f.setHours(21); // Cambia a 15/02/2020 21:30:10.999 (Devuelve
1581802210999)
f.setMinutes(00); // Cambia a 15/02/2020 21:00:10.999 (Devuelve
1581800410999)
f.setSeconds(3); // Cambia a 15/02/2020 21:00:03.999 (Devuelve
1581800403999)
f.setMilliseconds(79); // Cambia a 15/02/2020 21:00:03.079 (Devuelve
1581800403079)
f.setTime(872817240000); // Cambia a 29/08/1997 02:14:00.000 (Devuelve
872817240000)
```

Ten en cuenta que los mismos métodos anteriores, soportan varios parámetros, para hacer más cómodo su utilización y no tener que estar usándolos uno por uno. Así por ejemplo, se puede usar `setFullYear()` para cambiar año, año y mes, o año, mes y día:

Método	Descripción
`.setFullYear(y, m, d)`	Altera el año, mes y día de una fecha.
`.setMonth(m, d)`	Altera el mes y día de una fecha.
`.setHour(h, m, s, ms)`	Altera la hora, minutos, segundos y milisegundos.
`.setMinutes(m, s, ms)`	Altera los minutos, segundos y milisegundos.
`.setSeconds(s, ms)`	Altera los segundos y milisegundos.
`.setTime(ts)`	Establece una fecha a partir del **tiempo Unix** `ts`.

Por último, también tenemos a nuestra disposición el método setter `setTime(ts)`, que nos permite establecer una fecha a partir de un **tiempo Unix** `ts`. Es el equivalente a hacer un `new Date(ts)`.

Representación de fechas

Otro detalle más delicado dentro del trabajo con fechas es a la hora de mostrar una fecha en un formato específico. Podemos utilizar el formato por defecto que nos ofrece Javascript, pero lo más habitual es que queramos hacerlo en otro diferente, utilizando nuestra configuración horaria, abreviaturas u otros detalles. En Javascript tenemos varios métodos (*muy limitados, eso sí*) para representar las fechas:

Método	Descripción
Formato por defecto	`Fri Aug 24 2018 00:23:31 GMT+0100`
`.toDateString()`	Devuelve formato sólo de fecha: `Fri Aug 24 2018`
`.toLocaleDateString()`	Idem al anterior, pero en el formato regional actual: `24/8/2018`

Método	Descripción
`.toTimeString()`	Devuelve formato sólo de hora: `00:23:24 GMT+0100` ...
`.toLocaleTimeString()`	Idem al anterior, pero en el formato regional actual: `0:26:37`
`.toISOString()`	Devuelve la fecha en el formato ISO 8601: `2018-08-23T23:27:29.380Z`
`.toJSON()`	Idem al anterior, pero asegurándose que será compatible con JSON.
`.toUTCString()`	Devuelve la fecha, utilizando UTC (*ver más adelante*).

El método `toISOString()` devuelve un formato que intenta ser el estándar a la hora de manejar fechas en Internet o documentos en general. Básicamente, una regla mnemotécnica es pensar que se escriben desde magnitudes más altas (*año*) hasta las magnitudes más bajas (*milisegundos*). La fecha y la hora se separa con una `T` y cada componente con un `-` o un `:` dependiendo de si es fecha o es hora. Los milisegundos se separan siempre con `.`. La `Z` del final indica que es una fecha `UTC`.

Existen algunas librerías interesantes que permiten trabajar muy fácilmente con la representación de fechas o incluso otros detalles. Lo comentaremos un poco más adelante.

UTC y zonas horarias

Para facilitar las cosas, hasta ahora hemos obviado un tema bastante importante: las **zonas horarias**. Probablemente te habrás percatado de que podría ser necesario trabajar y manejar **diferencias horarias**, porque por ejemplo necesitamos utilizar diferentes horas locales de distintos lugares del planeta. Para manejar esto, Javascript permite indicar también la diferencia horaria respecto al meridiano Greenwich, indicada como `GMT`.

Por ejemplo, podemos tener una fecha A `2018/01/01 15:30:00 GMT+0000`, y una fecha B `2018/01/02 18:30:00 GMT+0200`. La fecha A no tiene diferencia horaria (GMT+0000), sin embargo, la fecha B si la tiene (GMT+0200), por lo que en Javascript al hacer un `new Date(str)` de dicha fecha, se recalcula automáticamente y se guarda la fecha modificada y sin diferencia horaria. En este caso, se le restan 2 horas a la fecha B.

Es importante darse cuenta que, en Javascript, cuando usamos `new Date()` sin parámetros, obtendremos una fecha que **puede** incluir diferencias horarias:

```
const f = new Date(); // Mon Aug 27 2018 01:39:21 GMT+0100
f.getTimezoneOffset(); // -60 (1 hora menos)
```

Sin embargo, si hacemos uso de `new Date(str)`, pasándole una fecha como parámetro de texto, automáticamente se recalcula la diferencia horaria, sumando/restando las horas, y se obtiene finalmente la fecha con diferencia horaria **cero**:

```
const f = new Date("2018/01/30 23:15:30 GMT+0100");
f; // Tue Jan 30 2018 22:15:30 GMT+0000
```

En este caso, al recalcular y guardar la fecha/hora sin diferencias horarias, decimos que estamos utilizando el **Tiempo Universal Coordinado** o UTC. También podemos utilizar el método estático `Date.UTC()` donde le podemos pasar los parámetros de los componentes numéricos de la fecha, tal cómo lo hacemos en el constructor `new Date(y, m, d, h, min, s, ms)` anteriormente mencionado.

Además de lo anterior, recuerda que también tenemos todos los métodos **Getters** y **Setters** mencionados en su versión UTC. Por ejemplo, en el caso de `getHours()` tenemos una versión `getUTCHours()` que devuelve las horas de acuerdo al **UTC** (*Tiempo Universal Coordinado*).

Librerías para fechas

En muchos casos, el objeto de Javascript puede quedarse corto para el usuario, que echa mucho de menos algunas funciones específicas, mayor comodidad al trabajar con fechas, o que simplemente necesita un mayor control.

Existen múltiples librerías alternativas para trabajar con fechas, citamos algunas de ellas:

Librería	Descripción	GitHub
Moment.js	Manejo de fechas y horas en JS.	moment/moment
date-fns	Moderna librería de fechas.	date-fns/date-fns
Day.js	Librería inmutable (2KB) alternativa a Moment.js	iamkun/dayjs
js-joda	Librería inmutable de fechas para JS.	js-joda/js-joda
Datejs	Librería alternativa para trabajar con fechas.	datejs/Datejs
fecha	Librería ligera para formatear y parsear fechas.	taylorhakes/fecha

Con ellas, podrás trabajar con mayor comodidad o profundidad con fechas y horas en Javascript

Funciones

Una vez conocemos las bases de las funciones que hemos explicado en el tema de introducción funciones básicas, podemos continuar avanzando dentro del apartado de las funciones. En Javascript, las **funciones** son uno de los tipos de datos más importantes, ya que estamos continuamente utilizándolas a lo largo de nuestro código.

Y no, no me he equivocado ni he escrito mal el texto anterior; a continuación veremos que las funciones también pueden ser tipos de datos:

```
typeof function () {}; // 'function'
```

Creación de funciones

Hay varias formas de crear funciones en Javascript: por **declaración** (*la más usada por principiantes*), por **expresión** (*la más habitual en programadores con experiencia*) o mediante constructor de **objeto** (*no recomendada*):

Constructor	Descripción
`function nombre(p1, p2...) { }`	Crea una función mediante **declaración**.
`var nombre = function(p1, p2...) { }`	Crea una función mediante **expresión**.
`new Function(p1, p2..., code);`	Crea una función mediante un constructor de **objeto**.

Funciones por declaración

Probablemente, la forma más popular de estas tres, y a la que estaremos acostumbrados si venimos de otros lenguajes de programación, es la primera, a la **creación de funciones por declaración**. Esta forma permite declarar una función que existirá a lo largo de todo el código:

```
function saludar() {
  return "Hola";
}

saludar(); // 'Hola'
typeof saludar; // 'function'
```

De hecho, podríamos ejecutar la función `saludar()` incluso antes de haberla creado y funcionaría correctamente, ya que Javascript primero busca las declaraciones de funciones y luego procesa el resto del código.

Funciones por expresión

Sin embargo, en Javascript es muy habitual encontrarse códigos donde los programadores «guardan funciones» dentro de variables, para posteriormente «ejecutar dichas variables». Se trata de un enfoque diferente, creación de funciones por **expresión**, que fundamentalmente, hacen lo mismo con algunas diferencias:

```
// El segundo "saludar" (nombre de la función) se suele omitir: es
redundante
const saludo = function saludar() {
  return "Hola";
};

saludo(); // 'Hola'
```

Con este nuevo enfoque, estamos creando una función **en el interior de una variable**, lo que nos permitirá posteriormente ejecutar la variable (*como si fuera una función*). Observa que el nombre de la función (*en este ejemplo: saludar*) pasa a ser inútil, ya que si intentamos ejecutar `saludar()` nos dirá que no existe y si intentamos ejecutar `saludo()` funciona correctamente.

¿Qué ha pasado? Ahora el nombre de la función pasa a ser el nombre de la variable, mientras que el nombre de la función desaparece y se omite, dando paso a lo que se llaman las **funciones anónimas** (*o funciones lambda*).

Funciones como objetos

Como curiosidad, debes saber que se pueden declarar funciones como si fueran **objetos**. Sin embargo, es un enfoque que no se suele utilizar en producción. Simplemente es interesante saberlo para darse cuenta que en Javascript todo pueden ser objetos:

```
const saludar = new Function("return 'Hola';");

saludar(); // 'Hola'
```

Funciones anónimas

Las **funciones anónimas** o funciones lambda son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla:

```
// Función anónima "saludo"
const saludo = function () {
  return "Hola";
};

saludo; // ƒ () { return 'Hola'; }
saludo(); // 'Hola'
```

Observa que en la última línea del ejemplo anterior, estamos **ejecutando** la variable, es decir, ejecutando la función que contiene la variable. Sin embargo, en la línea anterior hacemos referencia a la variable (*sin ejecutarla, no hay paréntesis*) y nos devuelve la función en sí.

La diferencia fundamental entre las funciones por declaración y las funciones por expresión es que estas últimas sólo están disponibles a partir de la inicialización de la variable. Si «ejecutamos la variable» antes de declararla, nos dará un error.

Callbacks

Ahora que conocemos las **funciones anónimas**, podremos comprender más fácilmente como utilizar **callbacks** (*también llamadas funciones callback o retrollamadas*). A grandes rasgos, un **callback** (*llamada hacia atrás*) es pasar una **función B por parámetro** a una **función A**, de modo que la función A puede ejecutar esa función B de forma genérica desde su código, y nosotros podemos definirlas desde fuera de dicha función:

```
// fB = Función B
const fB = function () {
  console.log("Función B ejecutada.");
};

// fA = Función A
const fA = function (callback) {
  callback();
};

fA(fB);
```

Esto nos podría permitir crear varias funciones para utilizar a modo de callback y reutilizarlas posteriormente con diferentes propósitos. De hecho, los **callbacks** muchas veces son la primera estrategia que se suele utilizar en Javascript para trabajar la asincronía, uno de los temas que veremos más adelante:

```
// fB = Función B (callback)
const fB = function () {
  console.log("Función B ejecutada.");
};

// fError = Función Error (callback)
const fError = function () {
  console.error("Error");
};

// fA = Función A
const fA = function (callback, callbackError) {
  const n = ~~(Math.random() * 5);
  if (n > 2) callback();
  else callbackError();
};

fA(fB, fError); // Si ejecutamos varias veces, algunas darán error y
otras no
```

Viendo este ejemplo, podemos planear ejecutar la función `fA()` cambiando los callbacks según nos interese, sin necesidad de crear funciones con el mismo código repetido una y otra vez. Además, en el caso de que las funciones **callbacks** sean muy cortas, muchas veces utilizamos directamente la función anónima, sin necesidad de guardarla en una variable previamente:

```
// fA = Función A
const fA = function (callback, callbackError) {
  const n = ~~(Math.random() * 5);
  if (n > 2) callback();
  else callbackError();
};

fA(
  function () {
    console.log("Función B ejecutada.");
  },
  function () {
    console.error("Error");
  }
);
```

Aunque, como se puede ver, se suele evitar para facilitar la legibilidad del código, y sólo se utiliza en casos muy específicos donde estás seguro que no vas a reutilizar la función callback o no te interesa guardarla en una variable.

Funciones autoejecutables

Pueden existir casos en los que necesites crear una función y ejecutarla sobre la marcha. En Javascript es muy sencillo crear **funciones autoejecutables**. Básicamente, sólo tenemos que envolver entre paréntesis la función anónima en cuestión (*no necesitamos que tenga nombre, puesto que no la vamos a guardar*) y luego, ejecutarla:

```
// Función autoejecutable
(function () {
  console.log("Hola!!");
})();

// Función autoejecutable con parámetros
(function (name) {
  console.log(`¡Hola, ${name}!`);
})("Manz");
```

De hecho, también podemos utilizar parámetros en dichas funciones autoejecutables. Observa que sólo hay que pasar dichos parámetros al final de la función autoejecutable.

Ten en cuenta, que si la función autoejecutable devuelve algún valor con `return`, a diferencia de las **funciones por expresión**, en este caso lo que se almacena en la variable es el valor que devuelve la función autoejecutada:

```
const f = (function (name) {
  return `¡Hola, ${name}!`;
})("Manz");

f; // '¡Hola, Manz!`
typeof f; // 'string'
```

Clausuras

Las **clausuras** o cierres, es un concepto relacionado con las funciones y los ámbitos que suele costar comprender cuando se empieza en Javascript. Es importante tener las bases de funciones claras hasta este punto, lo que permitirá entender las bases de una clausura.

A grandes rasgos, en Javascript, una clausura o cierre se define como una función que «encierra» variables en su propio ámbito (*y que continúan existiendo aún habiendo terminado la función*). Por ejemplo, veamos el siguiente ejemplo:

```
// Clausura: Función incr()
const incr = (function () {
  let num = 0;
  return function () {
    num++;
    return num;
  };
})();

typeof incr; // 'function'
incr(); // 1
incr(); // 2
incr(); // 3
```

Tenemos una **función anónima** que es también una función autoejecutable. Aunque parece una función por expresión, no lo es, ya que la variable `incr` está guardando lo que devuelve la función anónima autoejecutable, que a su vez, es otra función diferente.

La «magia» de las clausuras es que en el interior de la función autoejecutable estamos creando una variable `num` que se guardará en el ámbito de dicha función, por lo tanto existirá con el valor declarado: `0`.

Por lo tanto, en la variable `incr` tenemos una función por expresión que además conoce el valor de una variable `num`, que sólo existe dentro de `incr`. Si nos fijamos en la función que devolvemos, lo que hace es incrementar el valor de `num` y devolverlo. Como la variable `incr` es una clausura y mantiene la variable en su propio ámbito, veremos que a medida que ejecutamos `incr()`, los valores de `num` (*que estamos devolviendo*) conservan su valor y se van incrementando.

Arrow functions

Las **Arrow functions**, funciones flecha o «fat arrow» son una forma corta de escribir funciones que aparece en Javascript a partir de **ECMAScript 6**. Básicamente, se trata de reemplazar eliminar la palabra `function` y añadir `=>` antes de abrir las llaves:

```
const func = function () {
  return "Función tradicional.";
};

const func = () => {
  return "Función flecha.";
};
```

Sin embargo, las **funciones flechas** tienen algunas ventajas a la hora de simplificar código bastante interesantes:

- Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves (*{}*).
- Además, en ese caso, automáticamente se hace un `return` de esa única línea, por lo que podemos omitir también el `return`.
- En el caso de que la función no tenga parámetros, se indica como en el ejemplo anterior: `() =>`.
- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo: `e =>`.
- En el caso de que la función tenga 2 ó más parámetros, se indican entre paréntesis: `(a, b) =>`.
- Si queremos devolver un objeto, que coincide con la sintaxis de las llaves, se puede englobar con paréntesis: `({name: 'Manz'})`.

Por lo tanto, el ejemplo anterior se puede simplificar aún más:

```
const func = () => "Función flecha."; // 0 parámetros: Devuelve
"Función flecha"
const func = (e) => e + 1; // 1 parámetro: Devuelve el valor de e + 1
const func = (a, b) => a + b; // 2 parámetros: Devuelve el valor de a
+ b
```

Las **funciones flecha** hacen que el código sea mucho más legible y claro de escribir, mejorando la productividad y la claridad a la hora de escribir código.

Ámbito léxico de this

Aunque aún no la hemos utilizado, una de las principales diferencias de las **funciones flecha** respecto a las funciones tradicionales, es el valor de la palabra clave `this`, que no siempre es la misma.

Por ejemplo, si utilizamos una función de forma global en nuestro programa, no notaremos ninguna diferencia:

```
// Si son funciones globales
const a = function () {
  console.log(this);
};
const b = () => {
  console.log(this);
};

a(); // Window
b(); // Window
```

Sin embargo, si utilizamos una función en el interior de un objeto, como suele ser el caso más habitual, si encontraremos diferencias. Observa que en la primera función, donde se utiliza una función tradicional, el `this` devuelve el objeto padre de la función.

Por otro lado, en la segunda función, donde se utiliza una función flecha, el `this` no devuelve el objeto padre de la función, sino que devuelve `Window`.

```
padre = {
  a: function () {
    console.log(this);
```

```
  },
  b: () => {
    console.log(this);
  },
};

padre.a(); // padre
padre.b(); // Window
```

Esta es una diferencia clave que hay que tener bien en cuenta a la hora de trabajar con las **funciones flecha**. Una buena práctica es utilizar funciones tradicionales como las funciones de primer nivel y, luego, en su interior o en callbacks, utilizar funciones flecha.

Arrays

A medida que trabajamos en nuestro código, se hace necesario agrupar valores en una misma variable, para representar conjuntos de datos con cierta relación entre sí. Para ello, tenemos la opción de crear **objetos**, o unas variables más sencillas llamadas **arrays**.

¿Qué es un array?

Un es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa en el array. En Javascript, se pueden definir de varias formas:

Constructor	Descripción
`new Array(len)`	Crea un array de `len` elementos .
`new Array(e1, e2...)`	Crea un array con ninguno o varios elementos.
`[e1, e2...]`	Simplemente, los elementos dentro de corchetes: `[]`. **Notación preferida**.

Por ejemplo, podríamos tener un array que en su primera posición tenemos el 'a', en la segunda el 'b' y en la tercera el 'c'. En Javascript, esto se crearía de esta forma:

```
// Forma tradicional
const array = new Array("a", "b", "c");

// Mediante literales (preferida)
const array = ["a", "b", "c"]; // Array con 3 elementos
const empty = []; // Array vacío (0 elementos)
const mixto = ["a", 5, true]; // Array mixto (string, number, boolean)
```

Al contrario que muchos otros lenguajes de programación, Javascript permite que se puedan realizar arrays de **tipo mixto**, no siendo obligatorio que todos los elementos sean del mismo tipo de dato (*en el ejemplo anterior,*).

OJO: Al crear un array con `new Array(num)`, si solo indicamos un parámetro y `num` es un número, Javascript creará un array de `num` elementos sin definir. Es decir, `a = new Array(3)` sería equivalente a `a = [undefined, undefined, undefined]`. Esto no ocurre con su equivalente, `a = [3]`, donde estamos creando un array con un único elemento: `3`.

Acceso a elementos

Al igual que los , saber el número elementos que tiene un array es muy sencillo. Sólo hay que acceder a la propiedad `.length`, que nos devolverá el número de elementos existentes en un array:

Método	Descripción
`.length`	Devuelve el número de elementos del array.
`[pos]`	Operador que devuelve el elemento número `pos` del array.

Por otro lado, si lo que queremos es acceder a un elemento específico del array, no hay más que utilizar el operador `[]`, al igual que hacemos con los para acceder a un carácter concreto. En este caso, accedemos a la posición del elemento que queremos recuperar sobre el array:

```
const array = ["a", "b", "c"];

array[0]; // 'a'
array[2]; // 'c'
array[5]; // undefined
```

Recuerda que las posiciones empiezan a contar desde `0` y que si intentamos acceder a una posición que no existe (*mayor del tamaño del array*), nos devolverá un .

Añadir o eliminar elementos

Existen varias formas de añadir elementos a un array existente. Veamos los métodos que podemos usar para ello:

Método	Descripción
`.push(obj1, obj2...)`	Añade uno o varios elementos al final del array. Devuelve tamaño del array.
`.pop()`	Elimina y devuelve el último elemento del array.
`.unshift(obj1, obj2...)`	Añade uno o varios elementos al inicio del array. Devuelve tamaño del array.
`.shift()`	Elimina y devuelve el primer elemento del array.
`.concat(obj1, obj2...)`	Concatena los elementos (o elementos de los arrays) pasados por parámetro.

En los arrays, Javascript proporciona métodos tanto para insertar o eliminar elementos **por el final** del array: `push()` y `pop()`, como para insertar o eliminar elementos **por el principio** del array: `unshift()` y `shift()`. Salvo por esto, funcionan exactamente igual.

El método de inserción, `push()` o `unshift()` inserta los elementos pasados por parámetro en el array y devuelve el tamaño actual que tiene el array después de la inserción. Por otro lado, los métodos de extracción, `pop()` o `shift()`, extraen y devuelven el elemento.

```
const array = ["a", "b", "c"]; // Array inicial
```

```
array.push("d"); // Devuelve 4. Ahora array = ['a', 'b', 'c', 'd']
array.pop(); // Devuelve 'd'. Ahora array = ['a', 'b', 'c']

array.unshift("Z"); // Devuelve 4. Ahora array = ['Z', 'a', 'b', 'c']
array.shift(); // Devuelve 'Z'. Ahora array = ['a', 'b', 'c']
```

Además, al igual que en los , tenemos el método `concat()`, que nos permite concatenar los elementos pasados por parámetro en un array. Se podría pensar que los métodos `.push()` y `concat()` funcionan de la misma forma, pero no es exactamente así. Veamos un ejemplo:

```
const array = [1, 2, 3];
array.push(4, 5, 6); // Devuelve 6. Ahora array = [1, 2, 3, 4, 5, 6]
array.push([7, 8, 9]); // Devuelve 7. Ahora array = [1, 2, 3, 4, 5, 6,
[7, 8, 9]]

const array = [1, 2, 3];
array = array.concat(4, 5, 6); // Devuelve 6. Ahora array = [1, 2, 3,
4, 5, 6]
array = array.concat([7, 8, 9]); // Devuelve 9. Ahora array = [1, 2,
3, 4, 5, 6, 7, 8, 9]
```

Observa un detalle muy importante. El método `concat()`, a diferencia de `push()`, no modifica el array sobre el cuál trabajamos y al que le añadimos los elementos, sino que simplemente lo devuelve. Al margen de esto, observa que en el caso de pasar un array por parámetro, `push()` lo inserta como un array, mientras que `concat()` inserta cada uno de sus elementos.

También hay que tener cuidado al utilizar el operador `+` con los arrays. A diferencia de lo que quizás puede parecer intuitivo, utilizando este operador no se añaden los elementos al array, sino que se convierten los arrays en string y luego se concatenan. Veremos más sobre estas **conversiones implícitas** en temas posteriores.

Creación de arrays

Existen métodos para crear pequeños arrays derivados de otras variables u objetos. Es el caso de los métodos `slice()` y `splice()`. Luego, también hablaremos del método `join()` y el método estático `Array.from()`:

Método	Descripción
`.slice(ini, end)`	Devuelve los elementos desde posición `ini` hasta `end` (excluído).
`.splice(ini, num)`	Elimina y devuelve `num` elementos desde posición `ini`.
`.splice(ini, num, o1, o2...)`	Idem. Además inserta `o1`, `o2`... en la posición `ini`.
`.join(sep)`	Une los elementos del array por `sep` en un .
`Array.from(o, f, thisVal)`	Crea un array a partir de `o` (algo similar a un array).

El método `slice()` devuelve los elementos del array desde la posición `ini` hasta la posición `end`, permitiendo crear un nuevo array más pequeño con ese grupo de elementos. Recuerda que las posiciones empiezan a contar desde `0`. En el caso de que no

se proporcione el parámetro `end`, se devuelven todos los elementos desde la posición `ini` hasta el final del array.

Por otro lado, ten en cuenta que el array sobre el que realizamos el método `slice()` no sufre ninguna modificación, sólo se devuelve por parámetro el array creado. Diferente es el caso del método `splice()`, el cuál realiza algo parecido a `slice()` pero con una gran diferencia: **modifica el array original**. En el método `splice()` el segundo parámetro `num` no es la posición final del subarray, sino el tamaño del array final, es decir, el número de elementos que se van a obtener desde la posición `ini`.

Por lo tanto, con el método `splice()`, devolvemos un array con los elementos desde la posición `ini` hasta la posición `ini+num`. El array original es modificado, ya que se eliminan los elementos desde la posición `ini` hasta la posición `ini+num`. Es posible también indicar una serie de parámetros opcionales después de los mencionados, que permitirán además de la extracción de elementos, **insertar dichos elementos** justo donde hicimos la extracción.

Veamos un ejemplo ilustrativo:

```
const array = ["a", "b", "c", "d", "e"];

// .slice() no modifica el array
array.slice(2, 4); // Devuelve ['c', 'd']. El array no se modifica.

// .splice() si modifica el array
array.splice(2, 2); // Devuelve ['c', 'd']. Ahora array = ['a', 'b',
'e']
array.splice(1, 0, "z", "x"); // Devuelve []. Ahora array = ['a', 'z',
'x', 'b', 'e']
```

A raíz de este último ejemplo, también podemos insertar elementos en una posición concreta del array de estas dos formas alternativas: utilizando `slice()` y `concat()` o utilizando `splice()` y una característica que veremos más adelante llamada **desestructuración**:

```
const a = [1, 2, 3, 8, 9, 10];
a.slice(0, 3).concat([4, 5, 6, 7], a.slice(3, 6)); // [1, 2, 3, 4, 5,
6, 7, 8, 9, 10]

const a = [1, 2, 3, 8, 9, 10];
a.splice(3, 0, ...[4, 5, 6, 7]); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

En ciertos casos, nos podría interesar reducir el tamaño de un array para quedarnos con sus primeros elementos y descartar el resto. Hay una forma muy sencilla y eficiente que es modificar directamente el tamaño del array mediante `.length`.

Por ejemplo, hacer un `a.length = 4` en un array de 8 elementos, reducirá el array a los primeros 4 elementos de una forma eficiente, ya que no crea un nuevo array, sino que reduce el tamaño del actual y descarta el resto de elementos.

Además, también tenemos otro método con el que es posible crear un a partir de un . Se trata del método `split()` que vimos en el tema de los . En este caso, el método `join()`

es su contrapartida. Con `join()` podemos crear un con todos los elementos del array, separándolo por el texto que le pasemos por parámetro:

```
const array = ["a", "b", "c"];

array.join("->"); // Devuelve 'a->b->c'
array.join("."); // Devuelve 'a.b.c'
"a.b.c".split("."); // Devuelve ['a', 'b', 'c']
"5.4.3.2.1".split("."); // Devuelve ['5', '4', '3', '2', '1']
```

Ten en cuenta que, como se puede ver en el último ejemplo, `split()` siempre devolverá los elementos como .

Por último, mencionar también el método estático `Array.from()`. Aunque ahora no le encontraremos mucha utilidad, nos resultará muy interesante más adelante. Este método se suele utilizar para convertir variables «parecidas» a los arrays (*pero que no son arrays*) en arrays reales. Este el caso de variables como (*que pueden actuar como array de strings*) o de nodos de un documento (*elementos del DOM, como veremos en temas posteriores*):

```
Array.from("hola"); // ['h', 'o', 'l', 'a']
Array.from(document.querySelector("body")); // [body.document]
```

De forma opcional, `Array.from(obj)` puede recibir dos parámetros más, al margen de `obj`: una función `f` y un parámetro `thisVal`. El funcionamiento de estos parámetros es similar al del método `.map()` que veremos en un tema posterior de **Array functions**.

Búsqueda y comprobación

Existen varios métodos para realizar ciertas comprobaciones con arrays:

Método	Descripción
`Array.isArray(obj)`	Comprueba si `obj` es un array. Devuelve `true` o `false`.
`.includes(obj, from)`	Comprueba si `obj` es uno de los elementos incluidos en el array.
`.indexOf(obj, from)`	Devuelve la posición de la primera aparición de `obj` desde `from`.
`.lastIndexOf(obj, from)`	Devuelve la posición de la última aparición de `obj` desde `from`.

El primero de ellos, `Array.isArray(obj)` se utiliza para comprobar si `obj` es un array o no, devolviendo un booleano. Los otros tres métodos funcionan exactamente igual que sus equivalentes en los . El método `includes()` comprueba si el elemento `obj` pasado por parámetro es uno de los elementos que incluye el array, partiendo desde la posición `from`. Si se omite `from`, se parte desde `0`.

```
const array = [5, 10, 15, 20, 25];

Array.isArray(array); // true
array.includes(10); // true
array.includes(10, 2); // false
```

```
array.indexOf(25); // 4
array.lastIndexOf(10, 0); // -1
```

Por otro lado, tenemos `indexOf()` y `lastIndexOf()` dos funciones que se utilizan para devolver la posición del elemento `obj` pasado por parámetro, empezando a buscar en la posición `from` (*o `0` si se omite*). El primer método, devuelve la primera aparición, mientras que el segundo método devuelve la última aparición.

Modificación de arrays

Es posible que tengamos un array específico al que queremos hacer ciertas modificaciones donde `slice()` y `splice()` se quedan cortos (*o resulta más cómodo utilizar los siguientes métodos*). Existen algunos métodos introducidos en **ECMAScript 6** que nos permiten crear una versión modificada de un array, mediante métodos como `copyWithin()` o `fill()`:

Método	Descripción
`.copyWithin(pos, ini, end)`	Devuelve , copiando en `pos` los ítems desde `ini` a `end`.
`.fill(obj, ini, end)`	Devuelve un relleno de `obj` desde `ini` hasta `end`.

El primero de ellos, `copyWithin(pos, ini, end)` nos permite crear una copia del array que alteraremos de la siguiente forma: en la posición `pos` copiaremos los elementos del propio array que aparecen desde la posición `ini` hasta la posición `end`. Es decir, desde la posición `0` hasta `pos` será exactamente igual, y de ahí en adelante, será una copia de los valores de la posición `ini` a la posición `end`. Veamos algunos ejemplos:

```
const array = ["a", "b", "c", "d", "e", "f"];

// Estos métodos modifican el array original
array.copyWithin(5, 0, 1); // Devuelve ['a', 'b', 'c', 'd', 'e', 'a']
array.copyWithin(3, 0, 3); // Devuelve ['a', 'b', 'c', 'a', 'b', 'c']
array.fill("Z", 0, 5); // Devuelve ['Z', 'Z', 'Z', 'Z', 'Z', 'c']
```

Por otro lado, el método `fill(obj, ini, end)` es mucho más sencillo. Se encarga de devolver una versión del array, rellenando con el elemento `obj` desde la posición `ini` hasta la posición `end`.

Ordenaciones

En Javascript, es muy habitual que tengamos arrays y queramos ordenar su contenido por diferentes criterios. En este apartado, vamos a ver los métodos `reverse()` y `sort()`, utiles para ordenar un array:

Método	Descripción
`.reverse()`	Invierte el orden de elementos del array.
`.sort()`	Ordena los elementos del array bajo un criterio de **ordenación alfabética.**
`.sort(func)`	Ordena los elementos del array bajo un criterio de ordenación `func`.

En primer lugar, el método `reverse()` cambia los elementos del array en orden inverso, es decir, si tenemos `[5, 4, 3]` lo modifica de modo que ahora tenemos `[3, 4, 5]`. Por otro lado, el método `sort()` realiza una ordenación (*por orden alfabético*) de los elementos del array:

```
const array = ["Alberto", "Ana", "Mauricio", "Bernardo", "Zoe"];

// Ojo, cada línea está modificando el array original
array.sort(); // ['Alberto', 'Ana', 'Bernardo', 'Mauricio', 'Zoe']
array.reverse(); // ['Zoe', 'Mauricio', 'Bernardo', 'Ana', 'Alberto']
```

Un detalle muy importante es que estos dos métodos **modifican el array original**, además de devolver el array modificado. Si no quieres que el array original cambie, asegurate de crear primero una copia del array, para así realizar la ordenación sobre esa copia y no sobre el original.

Sin embargo, la ordenación anterior se realizó sobre y todo fue bien. Veamos que ocurre si intentamos ordenar un array de números:

```
const array = [1, 8, 2, 32, 9, 7, 4];

array.sort(); // Devuelve [1, 2, 32, 4, 7, 8, 9], que NO es el
resultado deseado
```

Esto ocurre porque, al igual que en el ejemplo anterior, el tipo de ordenación que realiza `sort()` por defecto es una ordenación alfabética, mientras que en esta ocasión buscamos una **ordenación natural**, que es la que se suele utilizar con números. Esto se puede hacer en Javascript, pero requiere pasarle por parámetro al `sort()` lo que se llama una **función de comparación**.

Función de comparación

Como hemos visto, la ordenación que realiza `sort()` por defecto es siempre una ordenación alfabética. Sin embargo, podemos pasarle por parámetro lo que se conoce con los nombres de **función de ordenación** o **función de comparación**. Dicha función, lo que hace es establecer otro criterio de ordenación, en lugar del que tiene por defecto:

```
const array = [1, 8, 2, 32, 9, 7, 4];

// Función de comparación para ordenación natural
const fc = function (a, b) {
  return a > b;
};

array.sort(fc); // Devuelve [1, 2, 4, 7, 8, 9, 32], que SÍ es el
resultado deseado
```

Como se puede ver en el ejemplo anterior, creando la función de ordenación `fc` y pasándola por parámetro a `sort()`, le indicamos como debe hacer la ordenación y ahora si la realiza correctamente.

Si profundizamos en la tarea que realiza el `sort()`, lo que hace concretamente es analizar pares de elementos del array en cuestión. El primer elemento es `a` y el segundo

elemento es `b`. Por lo tanto, al pasarle la función de comparación `fc`, dicha función se encargará de, si devuelve `true` cambia el orden de `a` y `b`, si devuelve `false` los mantiene igual. Esto es lo que se conoce como el método de la burbuja, uno de los sistemas de ordenación más sencillos.

Obviamente, el usuario puede crear sus propias **funciones de comparación** con criterios específicos y personalizados, no sólo el que se muestra como ejemplo.

Array functions

En un tema posterior, veremos como desde Javascript es posible realizar bucles mediante las **array functions**, unos métodos especiales de los que permiten realizar bucles de una forma más legible, utilizando callbacks.