

Actividades de ampliación

- 4.37. Busca en la documentación oficial de JavaScript más información sobre el paso de parámetros por referencia y aquellos tipos de datos que funcionan así de forma nativa.
- 4.38. Profundiza en las funciones propias de arrays, conjuntos y mapas, que el lenguaje pone a disposición de los programadores.
- 4.39. Investiga si existe algún otro tipo de función en JavaScript que no se haya especificado en esta unidad.
- 4.40. Reflexiona y escribe una argumentación razonada de cinco problemas en los que no sería posible una solución iterativa en favor de una solución recursiva.
- 4.41. Busca diez funciones proporcionadas por JavaScript que utilicen callbacks como parámetros.
- 4.42. Profundiza sobre el funcionamiento de las clausuras y escribe un pequeño tutorial donde expliques su funcionamiento apoyándote en dos ejemplos que diseñes.
- 4.43. Investiga qué otras formas avanzadas existen de recorrer un array.

Enlaces web de interés

-  **Mozilla Developers** - <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions>
(website para desarrolladores)
-  **JavaScript.info** - <https://es.javascript.info/>
(sitio web dedicado al aprendizaje de JavaScript)
-  **Google for Education** - <https://edu.google.com/>
(sitio web dedicado al aprendizaje de las ciencias de la computación)
-  **Comunidad JavaScript** - <https://www.javascript.com/>
(recursos para principiantes)
-  **ECMAScript 2022** - <https://262.ecma-international.org/13.0/>
(última actualización de la especificación del lenguaje)
-  **W3Resource** - <https://www.w3resource.com/>
(recursos libres para programadores front-end)



Programación orientada a objetos con JavaScript

Objetivos

- Entender las ventajas del paradigma de la programación orientada a objetos.
- Analizar las características de la programación orientada a objetos.
- Estudiar cómo es la particular orientación a objetos de JavaScript.
- Adquirir destreza gestionando objetos, propiedades y métodos.
- Interiorizar el funcionamiento de los prototipos.
- Aprender a utilizar los objetos predefinidos más importantes del lenguaje.

Contenidos

- 5.1. Introducción a la programación orientada a objetos
- 5.2. Gestión de objetos
- 5.3. Prototipos
- 5.4. Objetos predefinidos del lenguaje

Introducción

La programación orientada a objetos lleva cosechando un gran éxito de adopción entre la comunidad de desarrolladores desde los años 90. Su éxito radica en la facilidad con la que es posible desarrollar aplicaciones y en las semejanzas que tienen los elementos del programa con el mundo real.

Desde el punto de vista de JavaScript, en el que prácticamente todo es un objeto, el estudio de este modelo de programación es probablemente la tarea más importante. Si en la unidad anterior se estudió el corazón de JavaScript, en esta se analizan los genes del lenguaje, la estructura identitaria que lo impregna todo.

Una vez asimilados todos los conceptos de esta unidad, se habrá aprendido el núcleo del lenguaje, todo lo demás será interesante de aprender, pero accesorio.

5.1. Introducción a la programación orientada a objetos

En última instancia, el desarrollo de aplicaciones web y, en general, toda la ingeniería del software lo que pretenden es resolver un problema del mundo real de una forma más o menos automática. Es decir, resolver una necesidad.

El problema que surge es que, con frecuencia, la solución alcanzada se ajusta más al sistema donde se va a explotar la aplicación que a la propia naturaleza del problema a resolver.

El mundo está plagado de objetos. Hay que pararse y observar durante un segundo para darse cuenta de que todo lo que se ve son objetos: un armario, un cuadro, un vehículo, una lámpara e, incluso, las personas. Todos pueden ser considerados objetos porque comparten dos elementos en común: tienen una serie de propiedades (rasgos, colores, marcas, medidas...) y también un conjunto de comportamientos (encender, caminar, frenar, abrir...).

Quizás, si se cambia el enfoque y se desarrollan los programas centrándose la solución en los objetos y en las relaciones que se establecen entre ellos, se consigan resultados más intuitivos, legibles y con menor coste de mantenimiento.

De esto trata, precisamente, la programación orientada a objetos (en adelante, POO): es un modelo de programación cuya pretensión es acercar todo lo posible el mundo real a un sistema informático.

5.1.1. ¿Qué es la POO?

La POO es uno de los paradigmas de programación más usados de los últimos 20 años, ya que aporta un nuevo enfoque para la resolución de problemas complejos que se plantean en la programación estructurada. Intenta cambiar el punto de vista procedural (los pasos a seguir para resolver el problema) por un punto de vista centrado en la propia naturaleza del problema y los elementos que intervienen (formato de los datos y sus interacciones).

Nota técnica

En el contexto del desarrollo de aplicaciones un paradigma de programación es un conjunto de métodos sistemáticos aplicables a todos los niveles de diseño de un programa informático. Existen múltiples modelos; la elección de uno u otro depende del objetivo que se pretende alcanzar y del camino más eficiente para lograrlo.

La idea central consiste en unificar en un mismo elemento, llamado **objeto**, tanto los datos como las funciones que lo manipulan, de manera que se alcanza un mayor grado de modularidad. Así, puede tomarse cada entidad de un problema y representarla como un objeto, definiendo sus propiedades (atributos) y sus métodos (acciones o funciones).

Por ejemplo, al desarrollar una aplicación para gestionar los procesos internos de una ITV (inspección técnica de vehículos), hay que definir un objeto llamado «vehículo», que podrá ser de distintos tipos (coche, moto, furgoneta, camión...), pero que compartirán una serie de propiedades (marca, color, dimensiones, cilindrada, número de bastidor, número de ruedas) y métodos (arrancar, acelerar, frenar, girar...).

El desarrollo del programa consistirá en establecer la relación de comunicación que se crea entre los distintos objetos que intervienen en el problema.

Lo primero será detallar cuáles son los rasgos más importantes que definen la POO y qué supone su ventaja competitiva con respecto a otros modelos de programación.

5.1.2. Características

Siempre ha existido cierta reticencia en un buen número de desarrolladores para considerar JavaScript como un lenguaje orientado a objetos, y no están exentos de argumentos. Es cierto que la implementación del modelo orientado a objetos de JavaScript está bastante alejada de otros lenguajes como Java, C++ o C#, que sí tienen una implementación completa del modelo.

Las características que debe tener un modelo para considerarse orientado a objetos son las siguientes y sirven para entender un poco mejor el origen de la polémica:

- **Abstracción.** La abstracción permite reducir la complejidad del código puesto que se aislan los componentes esenciales de aquellos que no lo son. Se trata de reunir en un único objeto todas aquellas propiedades y comportamientos comunes que forman parte de una generalidad de objetos. En definitiva, se trata de una característica que permite utilizar objetos sin conocer los detalles de su implementación, lo que facilita una mayor reutilización del código, y reduce considerablemente tanto los costes de desarrollo como de mantenimiento.
- **Encapsulamiento.** Es un mecanismo de protección que permite ocultar algunas propiedades y métodos de los objetos, de manera que solo se puedan cambiar por medio de las operaciones definidas para ese objeto.
- **Herencia.** Es el mecanismo por el que una clase (un modelo de objetos), permite heredar las características (propiedades y métodos) de otra clase. Esto facilita que



se puedan definir nuevas clases basadas en otras ya existentes, generando así una jerarquía de clases dentro de la aplicación. Si una clase hereda de otra obtiene sus propiedades y métodos pudiendo añadir otros nuevos específicos de esa nueva clase.

En la Figura 5.1 se puede ver cómo desde la clase **Vehículo** (que contiene solo propiedades y métodos comunes a una generalidad de vehículos) han heredado dos clases, **Coche** y **Moto**, cada una de ellas aportando sus propias propiedades y métodos únicos. De esta forma, cuando se crean objetos de clases heredadas, contienen las propiedades y métodos de ambas clases.

Esta especialización que conduce a una jerarquía de clases puede profundizarse tanto como sea necesario.

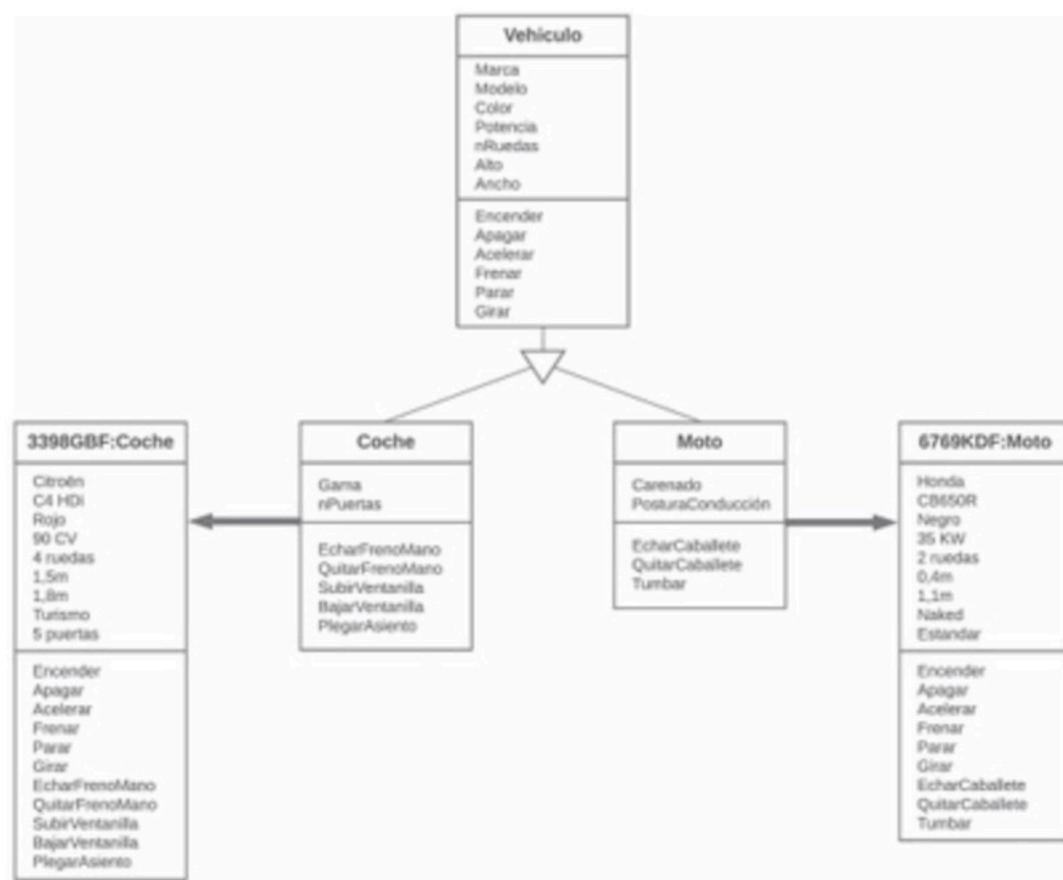


Figura 5.1. Diagrama de clases en una jerarquía de herencia.

- **Polimorfismo.** El polimorfismo es una interesante característica que permite que distintos tipos de objetos (objetos que heredan de clases distintas) tengan métodos con el mismo nombre, de manera que puedan realizar una misma acción de forma diferente. Por ejemplo, modelando una academia de formación se pueden tener una clase madre llamada **Miembro** (con un método llamado **Cobrar**) y dos clases derivadas llamadas **Profesor** y **Alumno**. De esta forma, tanto **Profesor** como **Alumno** podrían ejecutar el método **Cobrar** y obtener resultados diferentes: en el primer caso el salario del **Profesor** y en el segundo caso la beca del **Alumno**.

5.1.3. La POO en JavaScript

Tal y como se concibe de manera formal la POO, es necesario que el modelo de programación esté basado en clases. Pero JavaScript no es un lenguaje basado en clases, sino basado en prototipos.

Según la documentación de Mozilla: «JavaScript es un lenguaje basado en prototipos, toma el concepto de objeto prototípico, un objeto que se utiliza como una plantilla a partir de la cual se obtiene el **conjunto inicial** de propiedades de un nuevo objeto».

Toda aplicación JavaScript usa objetos y basa su lógica en las interacciones entre ellos. También es posible crear objetos sin utilizar clases. En su caso, utiliza prototipos que, igualmente, permiten heredar propiedades y métodos, y añadir otros específicos.

La realidad, por tanto, es que JavaScript trabaja de forma diferente los objetos en comparación a cómo lo hacen lenguajes orientados a objetos puros como C++ o Java, pero también es cierto que existen muchos lenguajes que, sin llegar al nivel de JavaScript, sí que relajan o reinterpretan las características vistas en el apartado anterior, y se siguen considerando orientados a objetos.

En cualquier caso, se trata de un debate superfluo, porque en la práctica ¿qué importa si se parte de clases o de prototipos cuando es posible beneficiarse de las mismas ventajas en ambos casos?

5.2. Gestión de objetos

Conocido el marco conceptual en el que moverse, es hora de empezar a escribir código. Gestionar objetos implica el conocimiento de importantísimos conceptos como son la distinción entre propiedades y métodos, qué es un objeto genérico **Object**, cómo se distingue entre objetos o qué operaciones se pueden realizar con ellos. Además, también se hace necesario conocer un nuevo lenguaje de intercambio de datos, muy utilizado en la actualidad en todas las aplicaciones web y que guarda una estrecha relación con los objetos de JavaScript. A continuación se resuelven todas estas incógnitas.

5.2.1. Propiedades y métodos

El uso de propiedades y métodos de un objeto, aunque no se haya analizado formalmente, se ha practicado con mucha frecuencia a lo largo de esta obra.

Acceso

Si, por ejemplo, se recuerda cómo se consigue mostrar en la consola el número de elementos de un array, se verá claramente cómo se accede a propiedades y métodos de un objeto:

```
let vector = [4,2,7,9];
console.log(vector.length);
```

En una sola línea de código, la segunda, se accede a la propiedad `length` del objeto `vector` y al método `log()` del objeto `console`. ¿Sorprendido? Con el siguiente código se puede ver cuál es el tipo de dato de `vector` y de `console`:

```
console.log(typeof(vector));
console.log(typeof(console));
```



Figura 5.2. Tipo de los objetos vector y console.

Así que, de forma genérica, se puede acceder a las propiedades de un objeto utilizando la notación punto (`.`), y también alternativamente la notación corchetes (`[]`). Estas instrucciones son equivalentes:

```
objeto.propiedad
objeto["propiedad"]
```

Por ejemplo, `vector.length` y `vector["length"]` son equivalentes.

Igualmente, puede hacerse lo mismo con los métodos, pero añadiendo los paréntesis que permiten pasar parámetros (no hay que olvidar que los métodos son funciones):

```
objeto.método(parámetros)
objeto["método"]([parámetros])
```

Por ejemplo, `console.log("Mensaje")` es equivalente a `console["log"]("Mensaje")`.

Instanceof

Es un operador que ayuda a comprobar el tipo de un objeto, es decir, el prototipo del que parte. Para usarlo hay que preguntarle si un objeto es de un prototipo concreto y devolverá `true` si lo es o `false` en caso contrario:

```
let vector = [4,2,7,9];
console.log(vector instanceof Array); // devuelve true
let conjunto = new Set();
console.log(conjunto instanceof Map); // devuelve false
```

5.2.2. Objeto Object

El objeto genérico `Object` representa a los **objetos literales** (o **instancias directas**), que son los objetos más sencillos que pueden crearse en JavaScript. No es necesario definirlos con una estructura completa previamente, sino que pueden irse construyendo al vuelo:

```
let notas = new Object();
notas.valores = [7,5,3,2,3,9,6];
notas.cantidad = notas.valores.length;
notas.media = notas.valores.reduce((a,b)=>a+b,0)/notas.cantidad;
notas.verMedia = function() {
```

```
    console.log(notas.media);
}
notas.verMedia(); // escribe 5
```

En este fragmento de código `new Object()` crea un objeto vacío que se va completando conforme va avanzando la ejecución. Así, se definen tres propiedades (`valores`, `cantidad` y `media`) que se van calculando sobre la marcha. Por último, se le añade un método, `verMedia()`, para mostrar la media de las notas.

Una representación alternativa podría ser aquella en la que se utilice la notación JSON, que se detallará más adelante:

```
let viaje={
    origen:"Granada",
    destino:"El Cairo",
    dias:8,
    precio:750,
    mostrar:function(){
        console.log(`${viaje.origen} / ${viaje.destino}`);
        console.log(`durante ${viaje.dias} días: EUR${viaje.precio}`);
    }
};
viaje.mostrar();
```

5.2.3. Objeto this

Es bastante frecuente que un método de un objeto necesite acceder a los datos almacenados en alguna de sus propiedades. Siguiendo con el ejemplo anterior, se podría reescribir el método `mostrar()` de este modo:

```
mostrar:function(){
    console.log(`${this.origen} / ${this.destino}`);
    console.log(`durante ${this.dias} días: EUR${this.precio}`);
}
```

Como se ve, se ha sustituido el objeto de referencia `viaje` por `this`. En tiempo de ejecución `this` valdrá `viaje`. Pero, entonces, ¿qué aporta `this`? Aporta seguridad, ya que si se utiliza directamente `viaje` el código no es confiable. Esto merece una explicación.

Si se hubiera asignado el objeto `viaje` a otro objeto, por ejemplo, `oferta (oferta = viaje)`, y posteriormente `viaje` modificara el estado de las propiedades, al utilizar el objeto `oferta` se ejecutaría código que contendría accesos a sus propiedades usando `viaje.propiedad`, lo cual es incorrecto en el contexto actual y podría generar errores (la estructura es la misma, pero con estados diferentes). A continuación se muestra un ejemplo:

```
let viaje={
    origen:"Granada",
    destino:"El Cairo",
    dias:8,
    precio:750,
    mostrar:function(){
        console.log(`${viaje.origen} / ${viaje.destino}`);
        console.log(`durante ${viaje.dias} días: EUR${viaje.precio}`);
    }
};
```

```

};

let oferta = viaje;
viaje = null;
oferta.mostrar();

② > Uncaught TypeError: Cannot read properties of null (reading 'origen')
    at Object.mostrar (poo.js:7:24)
    at poo.js:13:8
>

```

Figura 5.3. Error lanzado por la consola al intentar acceder a una propiedad.

En cambio, usando el objeto **this** en el método **mostrar()**, sí que puede acceder correctamente a sus propiedades:

```

let viaje={
    origen:"Granada",
    destino:"El Cairo",
    dias:8,
    precio:750,
    mostrar:function(){
        console.log(`${this.origen} / ${this.destino}`);
        console.log(`durante ${this.dias} días: EUR${this.precio}`);
    }
};
let oferta = viaje;
viaje = null;
oferta.mostrar();

```

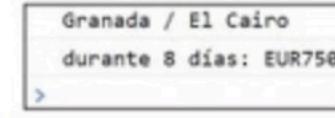


Figura 5.4. Salida correcta en consola.

Actividad resuelta 5.1

Primeros objetos

Crea un objeto denominado **usuario** que permita autenticar a la persona que quiere iniciar sesión en el sistema.

Solución

```

const usuario = {
    nombre: 'Antonio García',
    nombreUsuario: 'agar007',
    contraseña: 'agar007_pass',
    login: function(nombreUsuario, contraseña) {
        if (nombreUsuario === this.nombreUsuario && contraseña === this.contraseña) {
            console.log('Sesión iniciada con éxito');
        } else {
            console.log('Credenciales no válidas');
        }
    },
    usuario.login('agar007', 'agar007_pass');
    usuario.login('agar007', 'agar007pass');
}

```

5.2.4. Operaciones sobre objetos

Ha llegado el momento de poder operar con objetos. En los siguientes apartados se explican algunas de las operaciones fundamentales que pueden realizarse con ellos.

Constructores

Un constructor es un método especial que se utiliza para crear e inicializar de forma personalizada un objeto a partir de una clase.

Nota técnica

Las clases de JavaScript, introducidas en ECMAScript 2015, son una mejora sintáctica sobre la herencia basada en prototipos de JavaScript, no introduce un nuevo modelo de herencia orientada a objetos.

Solo puede haber un método llamado **constructor** en cada clase. El constructor se ejecuta inicializando las propiedades del objeto cuando se utiliza el operador **new**. Además, si la clase hereda de otra, se puede ejecutar el constructor de la clase madre invocando a **super()**.

```

class Viaje {
    origen = "Granada";
    destino = "El Cairo";
    dias = 8;
    precio = 750;
    constructor(or,des,di,pre) {
        this.origen = or;
        this.destino = des;
        this.dias = di;
        this.precio = pre;
    }
    mostrar(){
        console.log(`${this.origen} / ${this.destino}`);
        console.log(`durante ${this.dias} días: EUR${this.precio}`);
    }
}
let miViaje = new Viaje("Barcelona","Ibiza",2,112);
miViaje.mostrar();

```

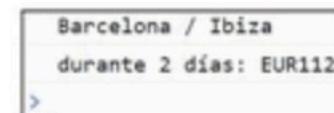


Figura 5.5. Constructor de la clase en funcionamiento.



Actividad propuesta 5.1

Constructores

Crea una clase para modelar un objeto «teléfono móvil» que tenga al menos estas propiedades: CPU, RAM, Almacenamiento, Ancho, Alto y numCamaras. Añade también un método llamado `toString()` que muestre en pantalla la información del objeto creado. Crea cuatro objetos con distintos números de parámetros en la creación y muestra en pantalla la información de cada objeto.

Actividad resuelta 5.2

Herencia y polimorfismo

Crea una jerarquía de clases y úsalas en un programa para exemplificar los conceptos de herencia y polimorfismo.

Solución

```
class Miembro {
    nombre = "nombre apellido1 apellido2";
    alta = "01/01/2022";
    estado = "vigente";
    constructor(nombre,alta,estado){
        this.nombre = nombre;
        this.alta = alta;
        this.estado = estado;
    }
    cobrar() {console.log(`El Miembro ${this.nombre} ha cobrado`);}
}

class Profesor extends Miembro {
    nAlumnos = 0;
    constructor(nombre,alta,estado,nAlumnos){
        super(nombre,alta,estado);
        this.nAlumnos = nAlumnos;
    }
    cobrar() {console.log(`El Profesor ${this.nombre} ha cobrado`);}
}

class Alumno extends Miembro {
    nAsignaturas = 0;
    constructor(nombre,alta,estado,nAsignaturas){
        super(nombre,alta,estado);
        this.nAsignaturas = nAsignaturas;
    }
    cobrar() {console.log(`El Alumno ${this.nombre} ha cobrado`);}
}

let unMiembro = new Miembro("Pepe Ruiz","12/02/2021","finalizado");
unMiembro.cobrar();
let unProfesor = new Profesor("Samuel Orta","25/06/2021","finalizado",30);
unProfesor.cobrar();
let unAlumno = new Alumno("Elena Sánchez","06/03/2022","finalizado",11);
unAlumno.cobrar();
```

Recorridos

Para recorrer un objeto se recomienda utilizar el bucle `for..in`, pero con cuidado porque se dan algunas circunstancias inesperadas, que se ven a continuación.

El bucle va a iterar sobre las propiedades del objeto:

```
let miViaje = new Viaje("Barcelona","Ibiza",2,112);
for (elemento in miViaje) {
    console.log(elemento);
}
```

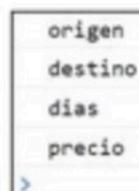


Figura 5.6. Salida del recorrido sobre las propiedades del objeto.

Pero también lo hará sobre las propiedades que herede en su cadena de prototipos, sacando primero las propiedades de los prototipos más cercanos. Si se quiere prevenir este comportamiento para que solo acceda a las propiedades del objeto en sí, y no de sus prototipos, puede utilizarse `getOwnPropertyNames()`.

```
for (elemento of Object.getOwnPropertyNames(miViaje)) {
    console.log(elemento);
}
```

No obstante, la iteración entre propiedades se hace de forma arbitraria, por lo cual no se recomienda que se modifiquen las propiedades de un objeto al mismo tiempo que se itera sobre ellas. No hay garantía de que se visite una propiedad que se ha modificado, e incluso podría intentarse visitar una propiedad eliminada dentro del propio recorrido.

Actividad propuesta 5.2

Recorridos con herencia

Utilizando los recorridos que se acaban de estudiar muestra todas las propiedades de los tres objetos creados en la solución de la Actividad resuelta 5.2.

Borrados

La eliminación de propiedades sí que es una operación bastante simple que no trae sorpresas. Tan solo hay que usar el operador `delete` sobre una propiedad de un objeto concreto:

```
let miViaje = new Viaje("Barcelona","Ibiza",2,112);
for (elemento of Object.getOwnPropertyNames(miViaje)) {
    console.log(elemento);
```

```

}
delete miViaje.precio;
delete miViaje.dias;
console.log("Eliminadas las propiedades precio y dias");
for (elemento of Object.getOwnPropertyNames(miViaje)) {
    console.log(elemento);
}

```

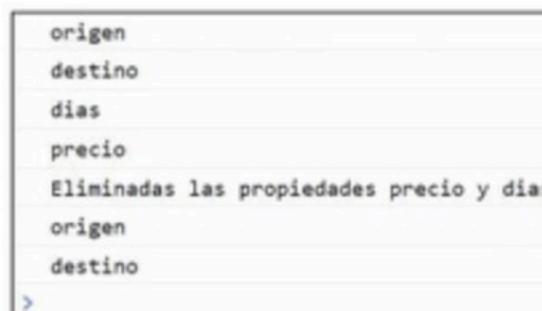


Figura 5.7. Evidencias del funcionamiento del operador delete.

5.2.5. JSON

JSON (*JavaScript Object Notation*) es un formato de representación basado en texto (ficheros .json) que utiliza la misma sintaxis de objetos literales de JavaScript, por lo que se utiliza con mucha frecuencia para el intercambio de datos en aplicaciones web.

No solo comparten sintaxis, sino que JavaScript incorpora un objeto llamado **JSON** que permite trabajar de forma muy eficiente con ese tipo de cadenas. Las cadenas de texto JSON deben ser convertidas previamente cuando se quiera acceder a sus datos a través de JavaScript.

Además, es importante saber que en JSON:

- Solo es posible definir propiedades, no métodos.
- Las propiedades deben ir entre comillas dobles.
- **stringify** convierte un objeto JavaScript a una cadena en formato JSON:

```

const receta={
    nombre:"Templada de Pulpo",
    tElaboración:30,
    formato: {
        tapa:"1 persona",
        media:"2 personas",
        racion:"4 personas"
    },
    ingredientes:['pulpo','patata','mayonesa','huevo','pimentón']
};
console.log(JSON.stringify(receta));
console.log(typeof(JSON.stringify(receta)));

```

```

{"nombre":"Templada de Pulpo","tElaboración":30,"formato":
{"tapa":"1 persona","media":"2 personas","racion":"4
personas"},"ingredientes":
["pulpo","patata","mayonesa","huevo","pimentón"]}
string
>

```

Figura 5.8. Salida de la conversión a cadena de un objeto JSON.

- **parse** devuelve el objeto equivalente en JavaScript desde una cadena JSON:

```

let cadena = new String('{"cancion":"Fuego","grupo":"Vetusta Morla"}');
console.log(JSON.parse(cadena));
let objeto = JSON.parse(cadena);
for (elemento in objeto) {
    console.log(elemento);
}

```

```

▼ {cancion: 'Fuego', grupo: 'Vetusta Morla'}
  cancion: "Fuego"
  grupo: "Vetusta Morla"
  ► [[Prototype]]: Object
  cancion
  grupo
>

```

Figura 5.9. Salida de la conversión en objeto de una cadena JSON.

Actividad propuesta 5.3

Trabajando con JSON

Busca en internet algún fichero JSON que forme parte de la API de algún servicio, cópialo en un **string**, luego conviértelo a un objeto de tipo JSON y realiza un recorrido para mostrar en la consola su contenido.

5.3. Prototipos

Repasando lo ya visto del modelo de objetos de JavaScript, se ha citado en varias ocasiones el concepto de prototipo. Hay que acudir al meollo de la cuestión para entender el verdadero ADN del lenguaje.

5.3.1. Una herencia especial

La gran mayoría de los lenguajes de programación orientados a objetos parte de la idea de clase, un modelo para crear objetos, que puede extenderse y, mediante especialización, ir construyendo una jerarquía de clases que van heredando unas de otras.

JavaScript, como ya se avanzó, no utiliza este esquema conceptual. En este lenguaje todos los objetos proceden de un prototipo (conjunto de propiedades y métodos comunes). Además, cuando en un esquema de clases puro se crean objetos, el código de los métodos se copia a los objetos de esa clase. En JavaScript, por el contrario, se enlaza con su prototipo. La ventaja de esta estrategia es que el prototipo se puede modificar al vuelo, y los objetos que lo enlazan estarán actualizados en tiempo real.

Los prototipos son, en definitiva, un mecanismo por el que los objetos de JavaScript heredan características entre sí.

5.3.2. Uso de prototipos

Todos los objetos en JavaScript se crean a partir de esas plantillas especiales llamadas prototipos. Para ser exactos, las propiedades y los métodos son definidos en la propiedad **prototype**, que reside en la función constructora del objeto, no en la instancia misma del objeto.

En JavaScript, por tanto, se crea un enlace entre la instancia del objeto y su prototipo (su propiedad **__proto__**, la cual es derivada de la propiedad **prototype** sobre el constructor), y las propiedades y los métodos son encontrados recorriendo la cadena de prototipos.

Con un ejemplo se verá un poco mejor cómo funciona el prototipo:

```
function Viaje(origen,destino,dias,precio) {
    this.origen = origen;
    this.destino = destino;
    this.dias = dias;
    this.precio = precio;
    this.mostrar = function(){
        console.log(`{this.origen} / {this.destino}`);
        console.log(`durante {this.dias} días: EUR${this.precio}`);
    };
}

let viaje1 = new Viaje("Barcelona","Ibiza",2,112);
console.log(viaje1);
```

Al acudir a la salida de la consola y desplegar el contenido del prototipo, se verán todas las propiedades y métodos que oferta (Figura 5.10).

El método **valueOf**, por ejemplo, retorna el valor del objeto sobre el que se llama. ¿Qué ocurriría si el objeto **Viaje** reescribiera el método **valueOf**?

- El navegador comprobará inicialmente si el objeto **viaje1** tiene un método **valueOf()** disponible en él.
- Si no lo tiene, el navegador comprobará si el objeto prototipo del objeto **viaje1** [es decir, el prototipo del constructor de **Viaje()**] tiene un método **valueOf()** disponible.
- Si tampoco lo tiene, entonces el navegador comprobará si **Object()** prototipo del objeto prototipo del constructor, tiene un método **valueOf()** disponible.

```
▼ Viaje {origen: 'Barcelona', destino: 'Ibiza', dias: 2, precio: 112, mostrar: f}
  destino: "Ibiza"
  dias: 2
  ▷ mostrar: f ()
  origen: "Barcelona"
  precio: 112
  ▷ [[Prototype]]: Object
    ▷ constructor: f Viaje(origen,destino,dias,precio)
    ▷ [[Prototype]]: Object
      ▷ constructor: f Object()
      ▷ hasOwnProperty: f hasOwnProperty()
      ▷ isPrototypeOf: f isPrototypeOf()
      ▷ propertyIsEnumerable: f propertyIsEnumerable()
      ▷ toLocaleString: f toLocaleString()
      ▷ toString: f toString()
      ▷ valueOf: f valueOf()
```

Figura 5.10. Información que proporciona la consola sobre el objeto **viaje1**.

Recuerda

Las propiedades y los métodos no se copian de un objeto a otro en la cadena del prototipo. Son accedidos subiendo por la cadena.

Además, se puede modificar cualquier prototipo operando con la propiedad **prototype** (evidentemente sobre el constructor). Si se hace sobre la conocida clase **Viaje**, se obtendría un prototipo estándar, puesto que nunca se ha modificado. Pero, se puede modificar y ver cómo dinámicamente asume la nueva situación:

```
let miViaje = new Viaje("Barcelona","Ibiza",2,112);
console.log(Viaje.prototype);
Viaje.prototype.costeDiario = function(){
    return this.precio/this.dias;
};
Viaje.prototype.descuento="20%";
console.log(Viaje.prototype);
```

```
▶ {constructor: f, mostrar: f}
▶ {descuento: '20%', costeDiario: f, constructor: f, mostrar: f}
>
```

Figura 5.11. Modificación dinámica del prototipo de **Viaje**.

Finalmente, recalcar una vez más que las propiedades y los métodos no se copian de un objeto a otro en la cadena del prototipo, sino que son accedidos subiendo por la cadena. Quizás, acceder de forma encadenada a la propiedad **__proto__** ayude a entender cómo conocen los navegadores quién es el objeto prototipo constructor del objeto:

```
let viaje1 = new Viaje("Barcelona","Ibiza",2,112);
console.log(viaje1.__proto__);
console.log(viaje1.__proto__.__proto__);
```

```

  ▼ {constructor: f} ①
    ► constructor: f Viaje(origen,destino,dias,precio)
    ► [[Prototype]]: Object

    {constructor: f, __defineGetter__: f, __defineSetter__: f,
     hasOwnProperty: f, __lookupGetter__: f, __proto__: Object}
    ► constructor: f Object()
  
```

Figura 5.12. Cadena de prototipos del objeto viaje1.

Como se observa, existe una cadena de prototipos a la que hacer referencia y que podría modificarse según los intereses.

5.4. Objetos predefinidos del lenguaje

Los objetos predefinidos del lenguaje son aquellos objetos que ya incorpora el lenguaje y que se pueden utilizar libremente con cualquier intérprete de JavaScript. En realidad, no es un concepto que resulte novedoso, puesto que ya se ha trabajado intensamente con muchos de ellos, como **String** (parcialmente), **Array**, **Set** o **Map**.

Esta sección descubre algunos otros objetos predefinidos que resultan de enorme interés por ser de utilidad en una generalidad de casos.

5.4.1. String

En la Unidad 2, al repasar los tipos de datos de JavaScript, se avanzaron algunos usos básicos de las cadenas de caracteres. En aquella ocasión se crearon como primitivas. Ahora, en cambio, se crearán de forma alternativa como objetos usando el constructor **String()**. Tanto las cadenas primitivas como las cadenas desde objeto se pueden usar indistintamente en la mayoría de las situaciones.

Las cadenas se pueden especificar encerrándolas entre comillas simples, 'cadena', dobles, "cadena", o invertidas, `cadena`, y su tratamiento será idéntico.

Las cadenas también se pueden comparar, tomando todas las precauciones (como se vio en la unidad anterior), teniendo en cuenta que el criterio por defecto de ordenación de los caracteres es el de Unicode.

En la Tabla 5.1 se recogen algunos de los métodos más frecuentemente utilizados con strings.

Tabla 5.1. Métodos útiles del objeto String

Método	Utilidad
charAt(<i>posición</i>)	Devuelve el carácter que ocupa esa <i>posición</i> .
charCodeAt(<i>posición</i>)	Variante que devuelve el código Unicode.

Método	Utilidad
toUpperCase()	Devuelve la cadena en mayúsculas.
toLowerCase()	Devuelve la cadena en minúsculas.
indexOf(texto)	Devuelve la posición del texto buscado.
lastIndexOf(texto)	Devuelve la posición de la última ocurrencia del texto.
endsWith(textoABuscar)	Devuelve true si el texto finaliza con lo buscado.
startsWith(textoABuscar)	Devuelve true si el texto empieza con lo buscado.
replace(txtAnt,txtNuevo)	Reemplaza el texto buscado por un nuevo texto.
trim()	Elimina los espacios en blanco del inicio y del final.
slice(inicio,fin)	Extrae la cadena desde la posición <i>inicio</i> hasta la posición <i>fin</i> , sin incluir <i>fin</i> .
substr(inicio,n)	Extrae <i>n</i> caracteres desde la posición <i>inicio</i> .
split(delimitador)	Rompe un <i>string</i> usando un carácter <i>delimitador</i> y construye un <i>array</i> con las piezas generadas.

Además, el objeto también dispone de tres métodos estáticos, como se muestra en la Tabla 5.2.

Tabla 5.2. Métodos estáticos del objeto String

Método	Utilidad
String.fromCharCode(num1[,...])	Crea una cadena usando una secuencia de valores Unicode.
String.fromCodePoint(num1 [,...])	Crea una cadena utilizando la secuencia de puntos de código especificada.
String.raw()	Crea una cadena a partir de una plantilla literal sin formato: no escapa caracteres.

```

let cadena = " Bolonia ";
console.log("1." + cadena.charAt(1));
console.log("2." + cadena.toUpperCase());
console.log("3." + cadena.toLowerCase());
console.log("4." + cadena.indexOf("n"));
console.log("5." + cadena.lastIndexOf("o"));
console.log("6." + cadena.replace("B","C"));
console.log("7." + cadena.trim());
console.log("8." + cadena.slice(1,3));
console.log("9." + cadena.substr(1,3));
console.log("10." + cadena.split("o"));
  
```

1.8
2. BOLONIA
3. bolonia
4.5
5.4
6. Colonia
7.Bolonia
8.Bo
9.sol
10. ,o,1,mia

Figura 5.13. Resultado de aplicar algunos de los métodos más comunes.

Nota técnica

Los métodos **estáticos** son aquellos que deben ser llamados sin instanciar su clase. Por ejemplo, para usar un método estático del prototipo `String()` no puede crearse un `objeto = new String()` y hacer luego `objeto.metodo()`, sino que hay que invocarlo directamente como `String.metodo()`.

**5.4.2. Date**

Los objetos **Date** representan un momento fijo en el tiempo que puede representarse en numerosos formatos. Una fecha en JavaScript se especifica como el número en milisegundos que han transcurrido desde el 1 de enero de 1970, UTC.

Importante

Es importante destacar que mientras el valor de la hora en el núcleo del objeto **Date** está en UTC, los métodos básicos para recibir la fecha y la hora o sus derivados trabajan todos en la zona horaria local.

Un objeto **Date** puede crearse sin parámetros, pero también indicando su lista completa (año, mes, día, hora, minutos, segundos, milisegundos) o un número parcial de ellos. Si se especifica uno solo, se interpreta que es el número de milisegundos transcurridos desde el 1 de enero de 1970. Lo mejor es ver algunos ejemplos:

```
let fechaSinParametros = new Date();
let fechaTodosParametros = new Date(2022,8,17,13,59,49,0);
let fechaTresParametros = new Date(2022,8,17);
let fechaUnParametro = new Date(1000);
```

```
Sun Jul 17 2022 19:30:37 GMT+0200 (hora de verano de Europa central)
Sat Sep 17 2022 13:59:49 GMT+0200 (hora de verano de Europa central)
Sat Sep 17 2022 00:00:00 GMT+0200 (hora de verano de Europa central)
Thu Jan 01 1970 01:00:01 GMT+0100 (hora estándar de Europa central)
```

Figura 5.14. Salidas en consola tras variar el número de parámetros del constructor de Date.

El número de parámetros del constructor es un aspecto útil del objeto, pero sin duda la gran cantidad de métodos disponibles permite hacer cualquier cálculo imaginable con fechas. Se repasan algunos de los más utilizados en la Tabla 5.3.

También existen algunas versiones de estos métodos, que se han obviado por claridad, pero que trabajan con UTC o GMT.

Por último, el objeto **Date** también dispone de un conjunto de métodos estáticos (Tabla 5.4).

Tabla 5.3. Métodos más usados del objeto Date

Método	Utilidad
<code>getDate()</code>	Devuelve el día del mes de la fecha (de 1 a 31).
<code>getDay()</code>	Obtiene el día de la semana de la fecha (de 0 a 6).
<code>getFullYear()</code>	Obtiene el año (con cuatro dígitos).
<code>getHours()</code>	Obtiene la hora de la fecha (número de 0 a 23).
<code>getMilliseconds()</code>	Obtiene los milisegundos en la fecha actual.
<code>getMinutes()</code>	Obtiene los minutos de la fecha.
<code>getMonth()</code>	Obtiene el número del mes sabiendo que enero es 0.
<code>getSeconds()</code>	Obtiene los segundos de la fecha.
<code>getTime()</code>	Obtiene el valor en milisegundos de la fecha.
<code> setDate(día)</code>	Modifica el día de la fecha.
<code>setFullYear(año)</code>	Modifica el año de la fecha.
<code>setHours(hora)</code>	Modifica la hora de la fecha.
<code>setMilliseconds(milisegundos)</code>	Modifica los milisegundos de la fecha.
<code>setMinutes(minutos)</code>	Modifica los minutos de la fecha.
<code>setMonth(mes)</code>	Modifica el mes de la fecha.
<code>setSeconds(segundos)</code>	Modifica los segundos de la fecha.
<code>setTime(milisegundos)</code>	Establece la fecha a partir de un valor en milisegundos.
<code>toDateString()</code>	Convierte la fecha a un formato más amigable.
<code>toLocaleString([params])</code>	Muestra la fecha en texto en formato local.
<code>toLocaleDateString()</code>	Muestra la fecha sin la hora en formato local.
<code>toTimeString()</code>	Muestra la hora sin la fecha en formato local.
<code>toString()</code>	Muestra la fecha en texto al estilo JavaScript.
<code>toJSON()</code>	Muestra la fecha en formato JSON.

Tabla 5.4. Métodos estáticos del objeto Date

Método	Utilidad
<code>Date.now()</code>	Devuelve el valor numérico correspondiente al actual número de milisegundos transcurridos desde el 1 de enero de 1970, 00:00:00 UTC, ignorando los segundos intercalares.
<code>Date.parse(objeto)</code>	Transforma la cadena que representa una fecha y retorna el número de milisegundos transcurridos desde el 1 de enero de 1970, 00:00:00 UTC, ignorando los segundos intercalares.
<code>Date.UTC(año, mes, día, horas, minutos, segundos, milisegundos)</code>	Acepta los mismos parámetros de la forma extendida del constructor (por ejemplo, del 2 al 7) y retorna el número de milisegundos transcurridos desde el 1 de enero de 1970, 00:00:00 UTC, ignorando los segundos intercalares.

Actividad resuelta 5.3

Fecha española

Haciendo uso de un objeto **Date**, crea un objeto de una clase que construyas para saludar al usuario tras iniciar sesión y que indique la fecha y la hora actuales.

Solución

```
class saludoHorario{
    usuario = "usuario";
    constructor(usuario) {
        this.usuario = usuario;
    }
    muestraSaludo() {
        let fecha = new Date();
        let dia = fecha.getDate() + "/" + fecha.getMonth() + "/" + fecha.getFullYear();
        let hora = fecha.getHours() + ":" + fecha.getMinutes() + ":" + fecha.getSeconds();
        console.log(`Bienvenido de nuevo ${this.usuario}.`);
        console.log(`Login registrado el ${dia} a las ${hora}.`);
    }
}
const saludar = new saludoHorario("David");
saludar.muestraSaludo();
```

5.4.3. Math

El objeto predefinido **Math** es otro que goza de mucha popularidad por la cantidad de operaciones matemáticas de cierta complejidad que resuelve. No es un objeto de función, no se puede editar y, además, todas sus propiedades y métodos son estáticos.

Incluye algunas constantes matemáticas de uso común como las recogidas en la Tabla 5.5.

Tabla 5.5. Constantes matemáticas definidas en el objeto Math

Constante	Utilidad
Math.E	Constante de Euler, aproximadamente 2,718.
Math.LN10	Logaritmo natural de 10, aproximadamente 2,303.
Math.LN2	Logaritmo natural de 2, aproximadamente 0,693
Math.LOG10E	Logaritmo de E con base 10, aproximadamente 0,434
Math.LOG2E	Logaritmo de E con base 2, aproximadamente 1,443.
Math.PI	Ratio de la circunferencia con respecto a su diámetro, aproximadamente 3,14159.
Math.SQRT1_2	Raíz cuadrada de ½, aproximadamente 0,707.
Math.SQRT_2	Raíz cuadrada de 2, aproximadamente 1,414.

Recuerda

Mucha de la funcionalidad matemática que se aporta en esta sección tiene una precisión que depende de la implementación. Diferentes navegadores pueden dar un resultado distinto, e incluso el mismo motor de JavaScript en un sistema operativo o arquitectura diferente puede dar resultados distintos.

En la Tabla 5.6 se recogen algunos de los métodos más usados por la comunidad de programadores.

Tabla 5.6. Métodos más usados del objeto Math

Método	Utilidad
Math.abs(n)	Valor absoluto de <i>n</i> .
Math.acos(n)	Arcocoseno de <i>n</i> .
Math.asin(n)	Arcoseno de <i>n</i> .
Math.atan(n)	Arcotangente de <i>n</i> .
Math.ceil(n)	Redondea <i>n</i> (decimal) a su entero superior.
Math.cos(n)	Coseno de <i>n</i> .
Math.exp(n)	e^n .
Math.floor(n)	Redondea <i>n</i> (decimal) a su entero inferior.
Math.log(n)	Logaritmo decimal de <i>n</i> .
Math.max(a,b)	Devuelve el mayor de dos números, <i>a</i> y <i>b</i> .
Math.min(a,b)	Devuelve el menor de dos números, <i>a</i> y <i>b</i> .
Math.pow(a,b)	a^b .
Math.random()	Devuelve un número aleatorio entre 0 y 1.
Math.round(n)	Redondea <i>n</i> a su entero más próximo.
Math.sin(n)	Seno de <i>n</i> .
Math.sqrt(n)	Raíz cuadrada de <i>n</i> .
Math.tan(n)	Tangente de <i>n</i> .
Math.trunc(n)	Devuelve la parte entera de <i>n</i> , eliminando los decimales.

Importante

Se ha de tener en cuenta que las funciones trigonométricas devuelven los ángulos en radianes. La conversión de radianes a grados se puede hacer con **Math.PI / 180**. Y a la inversa, de grados a radianes: **Math.PI * 180**.



Actividad propuesta 5.4

Trabajando con métodos matemáticos

Crea una clase llamada **Trigonometría** y encapsula en su interior los métodos relacionados con esta área de las Matemáticas, de manera que puedas usar los métodos en español: **sin** por seno, **cos** por coseno, **tan** por tangente, **asin** por arcoseno, **acos** por arcocoseno y **atan** por arcotangente. Finalmente, instancia un objeto de la clase y utiliza cada método sacando resultados por consola.

5.4.4. Boolean

El objeto **Boolean** es un objeto contenedor para un valor booleano, un valor lógico **true** o **false**. Por tanto, no se deben confundir los valores booleanos primitivos **true** y **false**, con los valores **true** y **false** del objeto **Boolean**.

El valor pasado a **Boolean** como primer parámetro se convierte en un valor booleano, si es necesario. Si el valor se omite o tiene uno de estos valores: **0**, **-0**, **null**, **false**, **NaN**, **undefined**, **""**, el objeto tiene un valor inicial de **false**. Para todos los demás valores se obtendrá inicialmente un valor de **true**.

```
let b1 = new Boolean(NaN);
let b2 = new Boolean(undefined);
let b3 = new Boolean("");
let b4 = new Boolean([]);
let b5 = new Boolean("false");
```

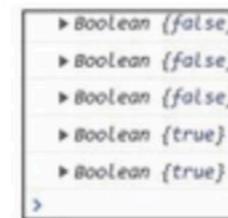


Figura 5.15. Resultados tras ejecutar el constructor de Boolean con diferentes valores.

Cuidado también al considerar estas dos expresiones porque no devuelven lo mismo y es un error común pensar lo contrario:

```
let logico1 = false;
let logico2 = new Boolean(false);
console.log(logico1);
console.log(logico2);
if (logico1)
    console.log("logico1 entra");
if (logico2)
    console.log("logico2 entra");
```

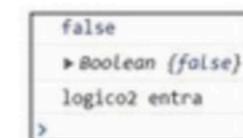


Figura 5.16. Un valor aparentemente false evaluándose como true.

La salida en la consola revela un comportamiento inesperado: aunque inicialmente ambas definiciones de las variables se muestran como **false**, no se evalúan como tal cuando se utilizan en expresiones condicionales. El objeto creado a partir de **Boolean** se ha evaluado como **true**.

La mejor regla que debe recordarse en relación con este objeto es no utilizar un objeto **Boolean** como sinónimo de un booleano primitivo.

5.4.5. Expresiones regulares

Las expresiones regulares, una utilidad que está presente en la mayoría de los lenguajes de programación, también cuentan con un objeto predefinido en JavaScript.

Conceptualmente una expresión regular es un patrón que se utiliza para buscar coincidencias en las cadenas de texto, de manera que puedan resolverse tareas frecuentes como la validación de datos.

Se pueden construir expresiones regulares de dos formas distintas:

1. Usando una expresión regular literal (las barras delimitan la expresión):

```
let erLiteral = /[0-9]/;
```

2. Llamando a la función constructora del objeto **RegExp**:

```
let erObjeto = new RegExp('[0-9]');
```

Por ejemplo, con las expresiones regulares anteriores serán validadas todas aquellas cadenas que contengan algún número, como por ejemplo "101 Dálmatas" o "875", pero no cadenas como "Faro" o "xMssT_pol#xhN".

La manera de comprobar si una cadena cumple con los criterios del patrón establecido en la expresión regular es a través del método **test()**, que recibe como parámetro la cadena a comprobar y devuelve **true** en caso de superar la validación, o **false** en caso contrario:

```
let erObjeto = new RegExp('[0-9]');
console.log(erObjeto.test("a"));
console.log(erObjeto.test("almamia"));
console.log(erObjeto.test("alma66Mia"));
console.log(erObjeto.test("987"));
```

Naturalmente, la salida de la pieza de código anterior será **false** **false** **true** **true**.

Debido a la altísima variabilidad de expresiones que se pueden indicar en una cadena de caracteres y las estrategias para validarlas, JavaScript ha desarrollado toda una sintaxis que facilita mucho su uso. A continuación se ven algunos de los elementos más comunes.

Modificador i

El símbolo i se indica cuando al validar letras del alfabeto no se desea distinguir entre mayúsculas y minúsculas:

```
let er = /a/;
console.log(er.test("pizza")); // Escribe true
console.log(er.test("TACO")); // Escribe false
let er2 = /a/i;
console.log(er2.test("pizza")); // Escribe true
console.log(er2.test("TACO")); // Escribe true
```

Modificador ^

El símbolo circunflejo, ^, fuerza que la cadena empiece por el carácter inmediatamente posterior:

```
let er = /^a/;
console.log(er.test("pizza")); // Escribe false
console.log(er.test("TACO")); // Escribe false
console.log(er.test("armario")); // Escribe true
```

Modificador \$

El símbolo del dólar, \$, fuerza a que la cadena termine por el carácter inmediatamente anterior:

```
let er = /pon$/;
console.log(er.test("ponderado")); // Escribe false
console.log(er.test("posicion")); // Escribe false
console.log(er.test("tapon")); // Escribe true
```

Modificador .

El símbolo del punto representa un carácter cualquiera:

```
let er = /ar.on/;
console.log(er.test("arcon")); // Escribe true
console.log(er.test("arpon")); // Escribe true
console.log(er.test("Aaron")); // Escribe false
```

Modificador []

Los símbolos de los corchetes establecen caracteres opcionales. La expresión la cumpliría cualquier cadena que contenga alguno de los elementos indicados entre corchetes:

```
let er = /[aeiou]/;
console.log(er.test("SOS")); // Escribe false
console.log(er.test("col")); // Escribe true
console.log(er.test("Pfff!")); // Escribe false
```

Modificador [^expresión]

El carácter circunflejo como primer elemento de unos corchetes indica un carácter no permitido. Por ejemplo, la siguiente expresión significa que solo se validarán las cadenas que no sean completamente numéricas:

```
let er = /^[^0-9]/;
console.log(er.test("cabo")); // Escribe true
console.log(er.test("526")); // Escribe false
console.log(er.test("bueno")); // Escribe true
console.log(er.test("p4ssw0rd")); // Escribe true
```

Modificadores de cardinalidad

Se trata de símbolos que permiten configurar repeticiones de expresiones (Tabla 5.7).

Tabla 5.7. Símbolos usados para modificar la cardinalidad en una expresión regular

Método	Utilidad
exp?	Halla ninguna o una vez el elemento exp. Por ejemplo, /a?sa?/ coincide con «as» en «pecas» y «asa» en «casados».
exp*	Si se usa inmediatamente después de cualquiera de los cuantificadores *, +, ?, o {}, hace que el cuantificador no sea maximalista (es decir, que coincida con el mínimo número de veces), a diferencia del predeterminado, que es maximalista (coincide con el máximo número de veces).
exp*	Concuerda cero o más veces con el elemento exp.
exp*	Por ejemplo, /ho*/ coincide con «muchoooo» en «Hace muchoooo calor» y «h» en «El vehículo está ardiendo», pero nada en «Para de ladrar».
exp+	Encuentra una o más veces el elemento exp, equivalente a {1,}.
exp+	Por ejemplo, /r+/ coincide con la letra «r» en «Brody» y con todas las letras «r» en «Brrrrrr! ».
exp{n}	Donde n es un número entero positivo, concuerda exactamente con n apariciones del elemento exp.
exp{n}	Por ejemplo, /r{2}/ no coincide con la «r» de «Brody», pero coincide con todas las «r» de «carro» y las dos primeras «r» en «Brrrrrr! ».
exp{n}	Donde n es un número entero positivo, concuerda con al menos n apariciones del elemento exp.
exp{n}	Por ejemplo, /a{2,}/ no coincide con las «a» en «maracaná», pero coincide con todas las «a» en «maamaa» y en «caaaaaaaraaaaamelo».
exp{m,n}	Donde n es 0 o un número entero positivo, m es un número entero positivo y m > n coincide con al menos n y como máximo m apariciones del elemento exp.
exp{m,n}	Por ejemplo, /e{1,3}/ no coincide con nada en «castaña», la «e» en «mesa», las dos «e» en «peero» y las tres primeras «e» en «eeeeeguro».

Actividad propuesta 5.5

Validación de etiquetas

Un conocido comercio para el que estás realizando su nuevo sitio web, te ha encargado que realices una primera validación de las referencias que aparecen en las etiquetas de sus productos. La etiqueta es de la forma 2022-xrFdS/25_9. Es decir, los 4 primeros caracteres deben ser numéricos, el siguiente debe ser un – los cinco siguientes serán combinaciones de letras minúsculas y mayúsculas, el siguiente carácter será la barra inclinada, los dos siguientes dos números, después un _ y por último un número.

Crea una clase llamada **Etiqueta**, que de momento tenga solo dos propiedades, nombre del artículo y referencia del artículo; y dos métodos, mostrar artículo y validar etiqueta. Luego crea un objeto y comprueba que se validan correctamente las etiquetas.

Modificador ()

Los símbolos de los paréntesis permiten agrupar expresiones, aumentando la complejidad del patrón. Por ejemplo, para validar una cadena del tipo «maria#5jorge#9», habría que indicarle que el patrón buscado es una cadena de cinco letras de la «a» a la «z», después el símbolo «#», después un número de «0» a «9», y después repetir lo anterior. Toda la expresión que va entre paréntesis es lo que debería repetirse:

```
let er = /([a-z]{5}#[0-9]){2}/;
console.log(er.test("maria#5jorge#9")); // Escribe true
console.log(er.test("maria#5jorge#")); // Escribe false
```

Modificador |

El símbolo de la barra vertical indica una opción, es decir, valida lo que está a su izquierda o lo que está a su derecha. Por ejemplo, para validar un teléfono móvil en España (nueve dígitos que deben comenzar por 6, 7 u 8) se puede recurrir a este código:

```
let er = /(6|7|8)([0-9]{8})/;
console.log(er.test("615833678")); // Escribe true
console.log(er.test("715833678")); // Escribe true
console.log(er.test("815833678")); // Escribe true
console.log(er.test("515833678")); // Escribe false
console.log(er.test("915833678")); // Escribe false
console.log(er.test("61583678")); // Escribe false
```

Modificadores abreviados

Se trata de un conjunto de símbolos a los que precede la barra invertida, que funcionan muy bien con Unicode y permiten escribir expresiones de una forma más ágil (Tabla 5.8).

Tabla 5.8. Símbolos usados para simplificar expresiones regulares

Símbolo	Utilidad
\d	Cualquier dígito numérico.
\D	Cualquier carácter salvo los dígitos numéricos.
\s	Espacio en blanco.
\S	Cualquier carácter salvo el espacio en blanco.
\w	Cualquier carácter alfanumérico: [a-zA-Z0-9].
\W	Cualquier carácter que no sea alfanumérico: [^a-zA-Z0-9].
\0	Carácter nulo.
\n	Carácter de nueva línea.
\t	Carácter tabulador.
\\\	El símbolo \.
\"	Comillas dobles.
\'	Comillas simples.
\c	Escapa el carácter c.
\ooo	Carácter Unicode empleando la notación octal.
\xff	Carácter ASCII empleando la notación hexadecimal.
\uffff	Carácter Unicode empleando la notación hexadecimal.

Actividad resuelta 5.4

Validación de correos electrónicos

Crea una función que haciendo uso de las expresiones regulares permita validar el formato de una dirección de correo electrónico. Recuerda que estas son las normas que validan un correo electrónico:

- El carácter @ es obligatorio; separa la primera parte (izquierda) de la segunda (derecha).
- La primera parte:
 - Acepta letras minúsculas y mayúsculas, caracteres numéricos y los caracteres especiales # * + & ‘ ! % @ ? { ^ } ”.
 - Acepta todos los caracteres punto (.) que se deseen, pero no puede ser ni el primer ni el último carácter, y tampoco pueden ir seguidos.
- La segunda parte acepta puntos, dígitos, guiones y letras.

Por ejemplo, hola@tu.casa.net es válido, pero mi.email.140dominio.com no lo es.

Solución

```
function validaEmail(email) {
    var regExp = /^[^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$/;
    if (regExp.test(email)) {
        console.log("Formato de email correcto");
    } else {
        console.log("Formato de email incorrecto");
    }
}
validaEmail("hola@tu.casa.net");
validaEmail("mi.email.140dominio.com");
```

Método exec()

El método **exec()** se utiliza para realizar una búsqueda sobre las coincidencias de una expresión regular en una cadena específica, devolviendo un **array** en caso de éxito o **null** en caso contrario. Básicamente, se trata de una alternativa más potente a **test()** y que muestra más información de las coincidencias.

La siguiente expresión:

```
let exreg = /sendero\s(arenoso).+?noche/ig;
let res = exreg.exec('El sendero arenoso de noche puede ser peligroso');
```

busca «sendero arenoso» seguido de «noche», ignorando los caracteres que encuentre en medio y, además, ignora mayúsculas y minúsculas. La información que contiene el objeto devuelto se observa en la Figura 5.17.

```
(2) ['sendero arenoso de noche', 'arenoso', index: 3, input: 'El
  sendero arenoso de noche puede ser peligroso', groups: undefined]
  ↳
  0: "sendero arenoso de noche"
  1: "arenoso"
  groups: undefined
  index: 3
  input: "El sendero arenoso de noche puede ser peligroso"
  length: 2
  ↳ [[Prototype]]: Array(0)
```

Figura 5.17. Toda la información proporcionada por la salida de exec().

La salida se interpreta de este modo:

- El elemento con índice cero es el primer texto encontrado que cumple la expresión regular.
- La propiedad **index** es un número que indica la primera posición en la que se encontró el texto.
- La propiedad **input** almacena el texto original donde se realizó la búsqueda.
- Los índices mayores que cero indican las coincidencias con las subcadenas buscadas en agrupaciones con paréntesis.

