LEARN JAVASCRIPT

BEGINNERS EDITION

Versión Española

A Complete Beginner's Guide to Learn JavaScript

Suman Kunwar

Tabla de contenido

Dedicatoria

Este libro está dedicado, con respeto y admiración, al espíritu de las computadoras y los lenguajes de programación en nuestro mundo.

> “El arte de programar es el arte de organizar la complejidad, de dominar la multitud y evitar su maldito caos de la manera más efectiva posible.”
>
> - Edsger Dijkstra

Derechos de autor

Aprenda JavaScript: Edición para principiantes Primera Edición

ASIN: B0C53J11V7

Prefacio

"Aprenda JavaScript: Edición para principiantes" es un libro que ofrece una exploración integral de JavaScript, posicionándolo como un lenguaje vital en el panorama digital en constante cambio. Centrándose en los fundamentos y la practicidad, este recurso está dirigido a todos los que deseen aprender el lenguaje de programación JavaScript.

El libro comienza cubriendo los aspectos fundamentales de JavaScript, avanzando poco a poco hacia técnicas más avanzadas. Aborda temas clave como variables, tipos de datos, estructuras de control, funciones, programación orientada a objetos, cierres (closures en inglés), promesas (promises en inglés) y sintaxis moderna. Cada capítulo se basa en el anterior, proporcionando una base sólida para los alumnos y facilitando la comprensión de conceptos complejos.

Una característica destacada de "Aprender JavaScript" es su enfoque práctico. El libro ofrece ejercicios prácticos, desafíos de codificación y problemas del mundo real que permiten a los lectores aplicar sus conocimientos y desarrollar habilidades esenciales. Al interactuar con ejemplos tangibles, los lectores ganan la confianza para abordar problemas comunes de desarrollo web y desbloquear el potencial de JavaScript para soluciones innovadoras.

Ideas complejas como cierres (closures en inglés) y programación asincrónica se desmitifican mediante explicaciones intuitivas y ejemplos prácticos. El énfasis en la claridad y la simplicidad permite a los estudiantes de todos los niveles captar y retener los conceptos clave de manera efectiva. El libro está estructurado en tres partes, y los primeros 14 capítulos profundizan en los conceptos centrales. Los cuatro capítulos siguientes profundizan en la utilización de JavaScript para la programación de navegadores web, mientras que los dos últimos capítulos cubren diversos temas y ofrecen ejercicios. La sección Varios explora temas y escenarios importantes relacionados con la programación JavaScript, seguidos de ejercicios para practicar.

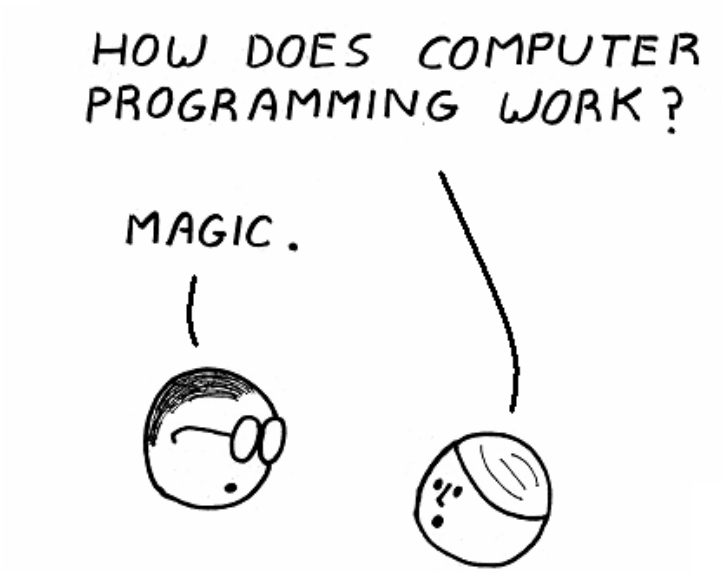
En conclusión, "Aprende JavaScript: Edición para principiantes" es una compañía esencial para quienes buscan dominar JavaScript y sobresalir en el desarrollo web. Con su cobertura integral, enfoque práctico, explicaciones claras, enfoque en aplicaciones del mundo real y compromiso con el aprendizaje continuo, este libro constituye un recurso valioso. Al sumergirse en su contenido, los lectores obtendrán las habilidades y el conocimiento necesarios para crear aplicaciones web dinámicas e interactivas, liberando todo su potencial como desarrolladores de JavaScript.

Capítulo 1

Introducción

Las computadoras son comunes en el mundo actual porque pueden realizar una amplia variedad de tareas de manera rápida y precisa. Se utilizan en muchas industrias diferentes, como los negocios, la atención médica, la educación y el entretenimiento, y se han convertido en una parte esencial de la vida diaria de muchas personas. Además de esto, también se utilizan para realizar cálculos científicos y matemáticos complejos, almacenar y procesar grandes cantidades de datos y comunicarse con personas de todo el mundo.

La programación implica crear un conjunto de instrucciones, llamado programa, para que las siga una computadora. Escribir un programa puede resultar tedioso y frustrante en ocasiones porque las computadoras son muy precisas y necesitan instrucciones específicas para completar las tareas.



Los lenguajes de programación son lenguajes artificiales que se utilizan para dar instrucciones a las computadoras. Se utilizan en la mayoría de las tareas de programación y se basan en la forma en que los humanos se comunican entre sí. Al igual que los lenguajes humanos, los lenguajes de programación permiten combinar palabras y frases para expresar nuevos conceptos. Es interesante observar que la forma más eficaz de comunicarse con las computadoras implica utilizar un lenguaje similar al lenguaje humano.

En el pasado, la primera forma de interactuar con las computadoras era a través de interfaces basados en lenguajes tales como BASIC y prompts de DOS. Estas han sido reemplazadas en gran medida por interfaces visuales, que son más fáciles de aprender pero ofrecen menos flexibilidad. Sin embargo, los lenguajes informáticos como *JavaScript* todavía están en uso y se pueden encontrar en los navegadores web modernos y en la mayoría de los dispositivos.

JavaScript (*JS para abreviar*) es el lenguaje de programación que se usa para crear interacción dinámica mientras desarrollamos páginas web, juegos, aplicaciones, e incluso servidores. JavaScript empezó en Netscape, un navegador web desarrollado en la decada de 1990, y hoy es uno de los lenguajes de programación más famosos y usados.

Inicialmente, fue creado para dar vida a las páginas web y solo podía ejecutarse en un navegador. Ahora se ejecuta en cualquier dispositivo que admita el motor JavaScript. Los objetos estándar como Array, Date y Math están disponibles en JavaScript, así como operadores, estructuras de control y declaraciones. *JavaScript del lado del cliente* y *JavaScript del lado del servidor* son las versiones extendidas del JavaScript principal.

- *JavaScript del lado del cliente* permite la mejora y manipulación de páginas web y navegadores de clientes. Las respuestas a eventos del usuario, como clics del mouse, entrada de formularios y navegación de páginas, son algunos de sus ejemplos.
- *JavaScript del lado del servidor* permite el acceso a servidores, bases de datos y sistemas de archivos.

JavaScript es un lenguaje interpretado. Mientras ejecuta Javascript, un intérprete interpreta cada línea y la ejecuta. El navegador moderno utiliza la tecnología Just In Time (JIT) para la compilación, que compila JavaScript en un código de bytes ejecutable.

> "LiveScript" fue el nombre inicial dado a JavaScript.

Código y qué hacer con él

Código son las instrucciones escritas que componen un programa. Aquí, muchos capítulos contienen mucho código y es importante leer y escribir código como parte del aprendizaje de programación. No se limite a leer rápidamente los ejemplos: léalos atentamente y trate de comprenderlos. Esto puede resultar difícil al principio, pero con la práctica mejorará. Lo mismo ocurre con los ejercicios: asegúrese de intentar escribir una solución antes de asumir que los comprende. También es útil intentar ejecutar las soluciones de los ejercicios en un intérprete de JavaScript, ya que esto le permitirá ver si su código funciona correctamente y puede animarlo a experimentar e ir más allá de los ejercicios.

Convenciones tipográficas

Aquí, el texto escrito en una fuente monoespaciada representa elementos de un programa. Puede ser un fragmento autónomo o una referencia a parte de un programa cercano. Los programas, como el que se muestra a continuación, se escriben de esta manera:

```
const numeros = [45, 4, 9, 16, 25];
let txt = "";
for (let x in numeros) {
  txt += numeros[x];
}
```

A veces, el resultado esperado de un programa se escribe después, precedido por dos barras con un *Resultado*, como este:

```
console.log(txt);

// Resultado: txt = '45491625'
```

Capítulo 2

Lo esencial

En este primer capítulo, aprenderemos lo esencial de la programación y el lenguaje Javascript.

Programar significa escribir código. Un libro se compone de capítulos, párrafos, oraciones, frases, palabras y finalmente puntuación y letras, así mismo un programa se puede dividir en componentes cada vez más pequeños. Por ahora, lo más importante es una sentencia. Una declaración es análoga a una oración en un libro. Por sí misma, tiene estructura y propósito, pero sin el contexto de las otras declaraciones a su alrededor, no es tan significativa.

Una declaración es casualmente (y comunmente) más conocida como una *línea de código*. Esto se debe a que las declaraciones tienden a escribirse en líneas individuales. Como tal, los programas se leen de arriba a abajo y de izquierda a derecha. Quizás te preguntes qué es el código (también llamado código fuente). Se trata de un término amplio que puede referirse a la totalidad del programa o a una parte más pequeña. Por lo tanto, una línea de código es simplemente una línea de su programa.

Aquí hay un ejemplo simple:

```
let hola = "Hola";
let mundo = "Mundo";

// El mensaje es igual a  "Hola Mundo"
let mensaje = hola + " " + mundo;
```

Este código puede ser ejecutado por otro programa llamado *intérprete* que leerá el código y ejecutará todas las declaraciones en el orden correcto.

Comentarios

Los comentarios son declaraciones que no serán ejecutandos por el intérprete, los comentarios son utilizados para marcar anotaciones para otros programadores o pequeñas descripciones de lo que el código hace, facilitando así que otros comprendan lo que hace su código. También se utilizan para desactivar temporalmente el código sin afectar al control de flujo del programa.

En JavaScript, los comentarios se pueden escribir de dos formas diferentes:

- *Comentarios de línea única*: Empiezan con dos barras diagonales (`//`) y continúa hasta el final de la línea. El intérprete de JavaScript ignora todo lo que sigue a las barras diagonales. Por ejemplo:

```
// Esto es un comentario, será ignorado por el intérprete
let a = "esto es una variable definida en una declaración";
```

- *Comentarios multilínea*: Comienzan con una barra diagonal y un asterisco (`/*`) y terminan con un asterisco y una barra diagonal (`*/`). El intérprete de JavaScript ignora todo lo que se encuentre entre los marcadores de apertura y cierre. Por ejemplo:

```
/*
Esto es un comentario multilínea,
será ignorado por el intérprete
*/
let a = "esto es una variable definida en una declaración";
```

Incluir comentarios en el código es esencial para mantener la calidad del código, habilitando la colaboración, y simplificando el proceso de depuración. Al proporcionar contexto y explicaciones para varias partes del programa, los comentarios facilitan la comprensión del código en el futuro. Por lo tanto, se considera una práctica beneficiosa incluir comentarios en el código.

Variables

El primer paso para comprender realmente la programación es mirar el álgebra. Si lo recuerdas de la escuela, el álgebra comienza escribiendo términos como los siguientes.

```
3 + 5 = 8
```

Comienzas a realizar cálculos cuando introduces una incógnita, por ejemplo, x a continuación:

```
3 + x = 8
```

Mover a quienes te rodean puede determinar x:

```
x = 8 - 3
-> x = 5
```

Cuando introduces más de uno, haces que tus términos sean más flexibles, estás utilizando variables:

```
x + y = 8
```

Puedes cambiar los valores de x e y y la fórmula aún puede ser verdadera:

```
x = 4
y = 4
```

o

```
x = 3
y = 5
```

Lo mismo ocurre con los lenguajes de programación. En programación, las variables son contenedores de valores que cambian. Las variables pueden contener todo tipo de valores y también los resultados de los cálculos. Las variables tienen un `nombre` y un `valor` separados por un signo igual (=). Sin embargo, es importante tener en cuenta que los diferentes lenguajes de programación tienen sus propias limitaciones y restricciones sobre lo que se puede utilizar como nombres de variables. Esto se debe a que ciertas palabras pueden reservarse para funciones u operaciones específicas dentro del idioma.

Veamos cómo funciona en Javascript. El siguiente código define dos variables, calcula el resultado de sumarlas y define este resultado como un valor de una tercera variable.

```
let x = 5;
let y = 6;
let resultado = x + y;
```

Hay ciertas pautas que se deben seguir al nombrar variables. Estas son

- Los nombres de las variables deben comenzar con una letra, un guión bajo (_) o un signo de dólar ($).
- Después del primer carácter podemos utilizar letras, números, guiones bajos o signos de dólar.
- JavaScript distingue entre letras mayúsculas y minúsculas (distingue entre mayúsculas y minúsculas), por lo que miVariable, MiVariable y MIVARIABLE son variables separadas.

- Para que su código sea fácil de leer y mantener, se recomienda utilizar nombres de variables descriptivos que reflejen con precisión su propósito.

Exercise

Defina una variable `x` que sea igual a 20.

```
let x =
```

Versión ES6

ECMAScript 2015 o ES2015 también conocido como ES6 es una actualización importante del lenguaje de programación JavaScript desde 2009. En ES6 tenemos tres formas de declarar variables.

```
var x = 5;
const y = 'Prueba';
let z = true;
```

Los tipos de declaración dependen del alcance. A diferencia de la palabra clave `var`, que define una variable global o localmente para una función completa independientemente del alcance del bloque, `let` le permite declarar variables cuyo alcance está limitado al bloque, declaración o expresión en el que se utilizan. Por ejemplo.

```
function varTest(){
    var x=1;
    if(true){
        var x=2; // la misma variable
        console.log(x); //2
    }
    console.log(x); //2
}

function letTest(){
    let x=1;
    if(true){
        let x=2;
        console.log(x); // 2
    }
    console.log(x); // 1
}
```

Las variables `const` son inmutables, lo que significa que no se permite reasignarlas.

```
const x = "¡hola!";
x = "adios"; // aquí ocurrirá un error
```

Tipos

Las computadoras son sofisticadas y pueden utilizar variables más complejas que solo números. Aquí es donde entran los tipos de variables. Las variables vienen en varios tipos y diferentes idiomas admiten diferentes tipos.

Los tipos más comunes son:

- **Number**: Los números pueden ser números enteros (por ejemplo, `1`, `-5`, `100`) o valores de punto flotante (por ejemplo, `3.14`, `-2.5`, `0.01`). JavaScript no tiene un tipo separado para números enteros y valores de punto flotante; los trata a ambos como números.
- **String**: Las cadenas son secuencias de caracteres, representadas por comillas simples (por ejemplo, `'hola'`) o comillas dobles (por ejemplo, `"mundo"`).
- **Boolean**: Los booleanos representan un valor verdadero o falso. Se pueden escribir como `true` o `false` (sin comillas).
- **Null**: El tipo nulo representa un valor nulo, lo que significa "sin valor". Se puede escribir como `null` (sin comillas).
- **Undefined**: El tipo indefinido representa un valor que no se ha establecido. Si una variable ha sido declarada, pero no se le ha asignado un valor, es `undefined`.
- **Object**: Un objeto es una colección de propiedades, cada una de las cuales tiene un nombre y un valor. Puede crear un objeto usando llaves (`{}`) y asignándole propiedades usando pares nombre-valor.
- **Array**: Una matriz (array en inglés) es un tipo especial de objeto que puede contener una colección de elementos. Puede crear una matriz usando corchetes (`[]`) y asignándole una lista de valores.
- **Function**: Una función es un bloque de código que se puede definir y luego llamar por su nombre. Las funciones pueden aceptar argumentos (entradas) y devolver un valor (salida). Puede crear una función utilizando la palabra clave `function`.

JavaScript es un lenguaje de "*tipado débil*", lo que significa que no es necesario declarar explícitamente de qué tipo de datos son las variables. Sólo necesita usar la palabra clave `var` para indicar que está declarando una variable, y el intérprete determinará qué tipo de datos está usando a partir del contexto y el uso de comillas.

Exercise

Declare tres variables e inicialícelas con los siguientes valores: `edad` como número, `nombre` como cadena y `estaCasada` como booleano.

```
let edad =
let nombre =
let estaCasada =
```

El operador `typeof` se usa para comprobar los tipos de datos de una variable.

```
typeof "Juan"                    // Devuelve "string"
typeof 3.14                      // Devuelve "number"
typeof NaN                       // Devuelve "number"
typeof false                     // Devuelve "boolean"
typeof [1,2,3,4]                 // Devuelve "object"
typeof {nombre:'Juan', edad:34}  // Devuelve "object"
typeof new Date()                // Devuelve "object"
typeof function () {}            // Devuelve "function"
typeof miCoche                   // Devuelve "undefined" *
typeof null                      // Devuelve "object
```

Los tipos de datos utilizados en JavaScript se pueden diferenciar en dos categorías según los valores que contienen.

Tipos de datos que pueden contener valores:

- `string`
- `number`
- `boolean`
- `object`
- `function`

`Object` , `Date` , `Array` , `String` , `Number` , y `Boolean` son los tipos de objetos disponibles en JavaScript.

Tipos de datos que no pueden contener valores:

- `null`
- `undefined`

Un valor de datos primitivo es un valor de datos simple sin propiedades ni métodos adicionales y no es un objeto. Son inmutables, lo que significa que no pueden modificarse. Hay 7 tipos de datos primitivos:

- string
- number
- bigint
- boolean
- undefined
- symbol
- null

Exercise

Declare una variable llamada `persona` e inicialícela con un objeto que contenga las siguientes propiedades: `edad` como un número, `nombre` como una cadena y `estaCasada` como un booleano.

```
let persona =
```

Igualdad

Al escribir un programa, con frecuencia necesitamos determinar la igualdad de variables en relación con otras variables. Esto se hace usando un operador de igualdad. El operador de igualdad más básico es el operador `==`. Este operador hace todo lo posible para determinar si dos variables son iguales, incluso si no son del mismo tipo.

Por ejemplo, supongamos:

```
let foo = 42;
let bar = 42;
let baz = "42";
let qux = "life";
```

`foo == bar` se evaluará como `true` y `baz == qux` se evaluará como `false`, como era de esperar. Sin embargo, `foo == baz` también se evaluará como `true` a pesar de que `foo` y `baz` sean tipos diferentes. Detrás de escena, el operador de igualdad `==` intenta forzar que sus operandos sean del mismo tipo antes de determinar su igualdad. Esto contrasta con el operador de igualdad `===`.

El operador de igualdad `===` determina que dos variables son iguales si son del mismo tipo *y* tienen el mismo valor. Con las mismas suposiciones que antes, esto significa que `foo === bar` aún se evaluará como `true`, pero `foo === baz` ahora se evaluará como `false`. `baz === qux` aún se evaluará como `false`.

Exercise

Use el operador `==` y `===`para comparar los valores de `str1` y `str2`.

```
let str1 = "hola";
let str2 = "HOLA";
let bool1 = true;
let bool2 = 1;
// comparar usando ==
let stringResult1 =
let boolResult1 =
// comparar usando ===
let stringResult2 =
let boolResult2 =
```

Capítulo 3

Números

JavaScript tiene **solo un tipo de número**: punto flotante de 64 bits. Es lo mismo que el `double` de Java. A diferencia de la mayoría de los otros lenguajes de programación, no existe un tipo entero separado, por lo que 1 y 1,0 tienen el mismo valor. Crear un número es fácil, se puede hacer como con cualquier otro tipo de variable usando la palabra clave `var`.

Los números se pueden crear desde un valor constante:

```
// Esto es un número de punto flotante (float):
let a = 1.2;

// Esto es un entero:
let b = 10;
```

O, desde el valor de otra variable:

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

La precisión de los números enteros es de hasta 15 dígitos y el número máximo es 17.

```
let x = 999999999999999;   // x será 999999999999999
let y = 9999999999999999;  // y será 10000000000000000
```

Las constantes numéricas se interpretan como hexadecimales si van precedidas de un `0x`.

```
let z = 0xFF; // 255
```

Math

El objeto `Math` permite realizar operaciones matemáticas en JavaScript. Es estático y no tiene constructor. Se pueden utilizar el método y las propiedades del objeto Math sin crear primero un objeto Math. Para acceder a su propiedad se puede utilizar *Math.property.* Algunas de las propiedades matemáticas se describen a continuación:

```
Math.E        // devuelve el número de Euler
Math.PI       // devuelve PI
Math.SQRT2    // devuelve la raíz cuadrada de 2
Math.SQRT1_2  // devuelve la raíz cuadrada de 1/2
Math.LN2      // devuelve el logaritmo natural de 2
Math.LN10     // devuelve el logaritmo natural de 10
Math.LOG2E    // devuelve el logaritmo en base 2 de E
Math.LOG10E   // devuelve el logaritmo en base 10 de E
```

Ejemplos de algunos de los métodos de Math son:

```
Math.pow(8, 2); // 64 (8 elevado al cuadrado)
Math.round(4.6); // 5
Math.ceil(4.9); // 5
Math.floor(4.9); // 4
Math.trunc(4.9); // 4
Math.sign(-4); // -1
Math.sqrt(64); // 8
Math.abs(-4.7); // 4.7
Math.sin(90 * Math.PI / 180);  // 1 (el seno de 90 grados)
Math.cos(0 * Math.PI / 180); // 1 (el coseno de 0 grados)
Math.min(0, 150, 30, 20, -8, -200); // -200
Math.max(0, 150, 30, 20, -8, -200); // 150
Math.random(); // 0.44763808380924375
Math.log(2); // 0.6931471805599453
Math.log2(8); // 3
Math.log10(1000); // 3
```

Para acceder a los metodos de Math, se pueden llamar a sus métodos directamente con argumentos cuando sea necesario.

Método	Descripción
`abs(x)`	Devuelve el valor absoluto de `x`
`acos(x)`	Devuelve el arcocoseno de `x` , en radianes
`acosh(x)`	Devuelve el arcocoseno hiperbólico de `x`
`asin(x)`	Devuelve el arcoseno de `x` , en radianes
`asinh(x)`	Devuelve el arcoseno hiperbólico de `x`
`atan(x)`	Devuelve el arcotangente de `x` como un valor numérico entre `-PI/2` y `PI/2` radianes
`atan2(y,x)`	Devuelve el arcotangente del cociente de sus argumentos.
`atanh(x)`	Devuelve el arcotangente hiperbólico de `x`
`crbt(x)`	Devuelve la raíz cubica de `x`
`ceil(x)`	Devuelve redondeado hacia arriba al número entero más cercano de `x`
`cos(x)`	Devuelve el coseno de `x` , en radianes
`cosh(x)`	Devuelve el coseno hiperbólico de `x`
`exp(x)`	Devuelve el valor exponencial de `x`
`floor(x)`	Devuelve redondeando hacia abajo al entero más cercano de `x`
`log(x)`	Devuelve el logaritmético natural de `x`
`max(x,y,z,... n)`	Devuelve el número con el valor más alto.
`min(x,y,z,... n)`	Devuelve el número con el valor más bajo.
`pow(x,y)`	Devuelve el valor de `x` elevado a `y`
`random()`	Devuelve un número aleatorio entre 0 y 1
`round(x)`	Redondea el número a la `x` más cercana
`sign(x)`	Devuelve -1 si `x` es negativo, 0 si es "null" o 1 si es positivo
`sin(x)`	Devuelve el seno de `x` , en radianes
`sinh(x)`	Devuelve el seno hiperbólico de `x`
`sqrt(x)`	Devuelve la raíz cuadrada de `x`
`tan(x)`	Devuelve la tangente del ángulo `x`
`tanh(x)`	Devuelve la tangente hiperbólica de `x`
`trunc(x)`	Devuelve la parte entera de un número (`x`)

Operadores básicos

Las operaciones matemáticas con números se pueden realizar utilizando algunos operadores básicos como:

- **Operador de suma (`+`)**: El operador de suma suma dos números. Por ejemplo:

```
console.log(1 + 2); // 3
console.log(1 + (-2)); // -1
```

- **Operador de resta (`-`)**: El operador de resta resta un número de otro. Por ejemplo:

```
console.log(3 - 2); // 1
console.log(3 - (-2)); // 5
```

- **Operador de multiplicación (`*`)**: El operador de multiplicación multiplica dos números. Por ejemplo:

```
console.log(2 * 3); // 6
console.log(2 * (-3)); // -6
```

- **Operador de división (`/`)**: El operador de división divide un número por otro. Por ejemplo:

```
console.log(6 / 2); // 3
console.log(6 / (-2)); // -3
```

- **Operador módulo (`%`)**: El operador módulo devuelve el resto de una operación de división. Por ejemplo:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

El intérprete de JavaScript funciona de izquierda a derecha. Se pueden usar paréntesis como en matemáticas para separar y agrupar expresiones:

`c = (a / b) + d`

> JavaScript utiliza el operador `+` tanto para la suma como para la concatenación. Los números se suman mientras que las cadenas se concatenan.

El término `NaN` es una palabra reservada que indica que un número no es un número legal, esto surge cuando realizamos aritmética con una cadena no numérica la cual dará como resultado `NaN` (Not a Number, no es un número, en español).

```
let x = 100 / "10";
```

El método `parseInt` analiza un valor como una cadena y devuelve el primer número entero.

```
parseInt("10"); // 10
parseInt("10.00"); // 10
parseInt("10.33"); // 10
parseInt("34 45 66"); // 34
parseInt(" 60 "); // 60
parseInt("40 years"); //40
parseInt("He was 40"); //NaN
```

En JavaScript, si calculamos un número fuera del mayor número posible, devuelve `Infinity` (infinito en español).

```
let x =  2 / 0; // Infinity
let y = -2 / 0; // -Infinity
```

Exercise

Utilice los operadores matemáticos +, -, *, / y % para realizar las siguientes operaciones en `num1` y `num2`.

```
let num1 = 10;
let num2 = 5;

// Sume num1 y num2.
let addResult =
// Reste num2 de num1.
let subtractResult =
// Multiplique num1 por num2.
let multiplyResult =
// Divida num1 por num2.
let divideResult =
// Encuentre el resto al dividir num1 por num2.
let reminderResult =
```

Operadores avanzados

Cuando los operadores se juntan sin paréntesis, el orden en el que se aplican está determinado por la *precedencia* de los operadores. La multiplicación `(*)` y la división `(/)` tienen mayor prioridad que la suma `(+)` y la resta `(-)`.

```
// Primero se hace la multiplicación y luego sigue la suma.
let x = 100 + 50 * 3; // 250
// con paréntesis las operaciones dentro del paréntesis se calculan primero
let y = (100 + 50) * 3; // 450
// las operaciones con las mismas precedencias se calculan de izquierda a derecha
let z = 100 / 50 * 3; // 6
```

Se pueden utilizar varios operadores matemáticos avanzados mientras se escribe el programa. Aquí hay una lista de algunos de los principales operadores matemáticos avanzados:

- **Operador Módulo (`%`)**: El operador de módulo devuelve el resto de una operación de división. Por ejemplo:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

- **Operador de exponenciación (`**`)**: El operador de exponenciación eleva un número a la potencia de otro número. Es un operador más nuevo y no es compatible con todos los navegadores, por lo que es posible que deba utilizar la función `Math.pow` en su lugar. Por ejemplo:

```
console.log(2 ** 3); // 8
console.log(3 ** 2); // 9
console.log(4 ** 3); // 64
```

- **Operador de incremento (`++`)**: El operador de incremento incrementa un número en uno. Se puede utilizar como prefijo (antes del operando) o como sufijo (después del operando). Por ejemplo:

```
let x = 1;
x++; // x es ahora 2
++x; // x es ahora 3
```

- **Operador de decremento (`--`)**: El operador de decremento disminuye un número en uno. Se puede utilizar como prefijo (antes del operando) o como sufijo (después del operando). Por ejemplo:

```
let y = 3;
y--; // y es ahora 2
--y; // y es ahora 1
```

- **Objeto Math**: El objeto `Math` es un objeto integrado en JavaScript que proporciona funciones y constantes matemáticas. Puede utilizar los métodos del objeto "Math" para realizar operaciones matemáticas avanzadas, como encontrar la raíz cuadrada de un número, calcular el seno de un número o generar un número aleatorio. Por ejemplo:

```
console.log(Math.sqrt(9)); // 3
console.log(Math.sin(0)); // 0
console.log(Math.random()); // un número aleatorio entre 0 y 1
```

Estos son sólo algunos ejemplos de las funciones y operadores matemáticos avanzados disponibles en JavaScript. Hay muchos más que puede usar para realizar operaciones matemáticas avanzadas mientras escribe un programa.

Exercise

Utilice los siguientes operadores avanzados para realizar operaciones en `num1` y `num2`.

```
let num1 = 10;
let num2 = 5;

// Use el operador ++ para incrementar el valor de num1.
const result1 =
// Use el operador -- para decrementar el valor de num2.
const result2 =
// Use el operador += para agregar num2 a num1.
const result3 =
// Use el operador -= para substraer num2 de num1.
const result4 =
```

Operador coalescente nulo '??'

El operador coalescente "nulo" devuelve el primer argumento si no es `null/undefined`, en caso contrario, el segundo. Está escrito como dos signos de interrogación `??`. El resultado de `x ?? y` es:

- Si `x` está definido, es `x`,
- Si `x` no está definida, es `y`.

Es una adición reciente al lenguaje y es posible que necesite polyfills para admitir navegadores antiguos.

Capítulo 4

Strings (Cadenas, en español)

Las cadenas (strings, en inglés) de JavaScript comparten muchas similitudes con implementaciones de cadenas de otros lenguajes de alto nivel. Representan mensajes y datos basados en texto. En este curso, cubriremos los conceptos básicos. Cómo crear nuevas cadenas y realizar operaciones comunes sobre ellas.

A continuación se muestra un ejemplo de una cadena:

```
"Hello World"
```

Los índices de cadenas tienen base cero, lo que significa que la posición inicial del primer carácter es "0" seguida de otros en orden incremental. Varios métodos son compatibles con cadenas y devuelven un nuevo valor. Estos métodos se describen a continuación.

Nombre	Descripción
`charAt()`	Devuelve el carácter en el índice especificado
`charCodeAt()`	Devuelve un carácter Unicode en el índice especificado
`concat()`	Devuelve dos o más cadenas combinadas
`constructor`	Devuelve la función constructora de la cadena.
`endsWith()`	Comprueba si una cadena termina con un valor específico
`fromCharCode()`	Devuelve valores Unicode como caracteres
`includes()`	Comprueba si una cadena contiene un valor especificado
`indexOf()`	Devuelve el índice de su primera aparición.
`lastIndexOf()`	Devuelve el índice de su última aparición.
`length`	Devuelve la longitud de la cadena.
`localeCompare()`	Compara dos cadenas con la configuración regional
`match()`	Compara una cadena con un valor o expresión regular
`prototype`	Se utiliza para agregar propiedades y métodos de un objeto.
`repeat()`	Devuelve una nueva cadena con el número de copias especificado
`replace()`	Devuelve una cadena con valores reemplazados por una expresión regular o una cadena con un valor
`search()`	Devuelve un índice basado en la coincidencia de una cadena con un valor o expresión regular
`slice()`	Devuelve una cadena que contiene parte de una cadena.
`split()`	Divide la cadena en una serie de subcadenas
`startsWith()`	Comprueba cadenas que comienzan con el carácter especificado
`substr()`	Extrae parte de la cadena, desde el índice inicial.
`substring()`	Extrae parte de la cadena, entre dos índices.
`toLocalLowerCase()`	Devuelve una cadena con caracteres en minúscula usando la configuración regional del host
`toLocalUpperCase()`	Devuelve una cadena con caracteres en mayúscula utilizando la configuración regional del host
`toLowerCase()`	Devuelve una cadena con caracteres en minúscula
`toString()`	Devuelve una cadena o un objeto de cadena como cadena
`toUpperCase()`	Devuelve una cadena con caracteres en mayúsculas
`trim()`	Devuelve una cadena con espacios en blanco eliminados
`trimEnd()`	Devuelve una cadena con espacios en blanco eliminados del final
`trimStart()`	Devuelve una cadena con espacios en blanco eliminados desde el inicio.
`valueOf()`	Devuelve el valor primitivo de una cadena o un objeto string.

Creación

Las cadenas se pueden definir encerrando el texto entre comillas simples o dobles:

```
// Se pueden utilizar comillas simples
let str = "Nuestra hermosa cadena";

// Comillas dobles también
let otherStr = "Otra bonita cadena";
```

En Javascript, las cadenas pueden contener caracteres UTF-8:

```
"中文 español English हिन्दी العربية português বাংলা русский 日本語 ਪੰਜਾਬੀ 한국어";
```

También puedes usar el constructor `String` para crear un objeto de cadena:

```
const stringObject = new String('Esto es una cadena');
```

Sin embargo, generalmente no se recomienda utilizar el constructor `String` para crear cadenas, ya que puede causar confusión entre las primitivas de cadena y los objetos de cadena. Normalmente es mejor utilizar cadenas literales para crear cadenas.

También puede utilizar literales de plantilla para crear cadenas. Los literales de plantilla son cadenas que están encerradas entre comillas invertidas (``) y pueden contener marcadores de posición para valores. Los marcadores de posición se indican con la sintaxis `${}` .

```
const nombre = 'Juan';
const mensaje = `¡Hola, ${nombre}!`;
```

Los literales de plantilla también pueden contener varias líneas y pueden incluir cualquier expresión dentro de los marcadores de posición.

Las cadenas no se pueden restar, multiplicar ni dividir.

Exercise

Utilice un literal de plantilla para crear una cadena que incluya los valores de `nombre` y `edad`. La cadena debe tener el siguiente formato: "Mi nombre es Juan y tengo 25 años".

```
let nombre = "Juan";
let edad = 25;

// Mi nombre es Juan y tengo 25 años.
let resultado =
```

Replace

El método `replace` nos permite reemplazar un caracter, una palabra, o una declaración dentro de una cadena. Por ejemplo.

```
let str = "¡Hola, mundo!";
let new_str = str.replace("Hola", "Hey");

console.log(new_str);

// Resultado: ¡Hey, mundo!
```

> Para reemplazar un valor en todas las instancias de una expresión regular se establece con un modificador `g` .

Busca una cadena para un valor o una expresión regular y devuelve una nueva cadena con los valores reemplazados. No cambia la cadena original. Veamos el ejemplo de reemplazo global que no distingue entre mayúsculas y minúsculas.

```
let texto = "El Señor Azul tiene un hogar azul y un auto azul.";
let resultado = texto.replace(/azul/gi, "rojo");

console.log(resultado);
//Resultado: El Señor rojo tiene un hogar rojo y un auto rojo.
```

Length

Es fácil en Javascript saber cuántos caracteres hay en una cadena usando la propiedad `length`. La propiedad `length` (en español, su significado es 'longitud') devuelve el número de caracteres de la cadena, incluidos espacios y caracteres especiales.

```
let size = "Nuestra hermosa cadena".length;
console.log(size);
// size: 17

let emptyStringSize = "".length
console.log(emptyStringSize);
// emptyStringSize: 0
```

La propiedad length de una cadena vacía es `0`.

> La propiedad `length` es una propiedad de solo lectura, así que no puede asignarle un nuevo valor.

Concatenación

La concatenación implica sumar dos o más cadenas, creando una cadena más grande que contiene los datos combinados de esas cadenas originales. La concatenación de una cadena agrega una o más cadenas a otra cadena. Esto se hace en JavaScript de las siguientes maneras.

- usando el operador `+`
- usando el método `concat()`
- usando el método del objeto Array `join()`
- usando el literal de plantilla (introducido en ES6)

El método de String `concat()` acepta la lista de cadenas como parámetros y devuelve una nueva cadena después de la concatenación, es decir, una combinación de todas las cadenas. Mientras que el método de Array `join()` se utiliza para concatenar todos los elementos presentes en una matriz convirtiéndolos en una sola cadena.

El literal de plantilla utiliza una comilla invertida `(``)` y proporciona una manera fácil de crear cadenas multilínea y realizar interpolación de cadenas. Se puede usar una expresión dentro de la comilla invertida usando el signo `$` y llaves `${expresión}` .

```
const icon = '👋';
// usando literal de plantilla
`hi ${icon}`;

// usando el método join()
['hi', icon].join(' ');

// usando el método concat()
''.concat('hi ', icon);

//  usando el operador +
'hi ' + icon;
// hi 👋
```

Capítulo 5

Lógica condicional

Una condición es una prueba para algo. Las condiciones son muy importantes para la programación, de varias maneras:

En primer lugar, se pueden utilizar condiciones para garantizar que su programa funcione, independientemente de los datos que le arroje para su procesamiento. Si confía ciegamente en los datos, se meterá en problemas y sus programas fallarán. Si pruebas que lo que quieres hacer es posible y tiene toda la información requerida en el formato correcto, eso no sucederá y tu programa será mucho más estable. Tomar tales precauciones también se conoce como programación defensiva.

La otra cosa que las condiciones pueden hacer por usted es permitir la ramificación. Es posible que haya encontrado diagramas de ramificación antes, por ejemplo, al completar un formulario. Básicamente, esto se refiere a ejecutar diferentes “ramas” (partes) de código, dependiendo de si se cumple o no la condición.

En este capítulo, aprenderemos las bases de la lógica condicional en JavaScript.

If

La condición más sencilla es una declaración if y su sintaxis es `if (condición) {haz esto...}`. La condición debe ser verdadera para que se ejecute el código dentro de las llaves. Por ejemplo, puede probar una cadena y establecer el valor de otra cadena dependiendo de su valor como se describe a continuación.

```
let pais = "Francia";
let clima;
let comida
let divisa;

if (pais === "Gran Bretaña") {
  clima = "horrible";
  comida = "relleno";
  divisa = "libra esterlina";
}

if (pais === "Francia") {
  clima = "lindo";
  comida = "impresionante, pero casi nunca vegetariana";
  divisa = "divertida, pequeña y colorida";
}

if (pais === "Alemania") {
  clima = "promedio";
  comida = "lo peor que he visto nunca";
  divisa = "divertida, pequeña y colorida";
}

let mensaje =
  "esto es " +
  pais +
  ", el clima es " +
  clima +
  ", la comida es " +
  comida +
  " y la " +
  "divisa es " +
  divisa;

console.log(mensaje);
// 'Esto es Francia, el clima es agradable, la comida es espectacular, pero casi nunca es vegetariana y la moneda es divertida
```

Las condiciones también se pueden anidar.

Else

También hay una cláusula `else` que se aplicará cuando la primera condición no sea verdadera. Esto es muy poderoso si desea reaccionar ante cualquier valor, pero seleccione uno en particular para recibir un tratamiento especial.

```
let paraguasObligatorio;

if (pais === "Inglaterra") {
  paraguasObligatorio = true;
} else {
  paraguasObligatorio = false;
}
```

La cláusula `else` se puede unir con otra `if`. Rehagamos el ejemplo del artículo anterior.

```
if (pais === "Inglaterra") {
  ...
} else if (pais === "Francia") {
  ...
} else if (pais === "Alemania") {
  ...
}
```

Exercise

A partir de los siguientes valores, escriba una declaración condicional que verifique si `num1` es mayor que `num2`. Si es así, asigne "num1 es mayor que num2" a la variable `resultado`. Si no es así, asigne "num1 es menor o igual que num2".

```
let num1 = 10;
let num2 = 5;
let resultado;

// comprobar si num1 es mayor que num2
if( condition ) {

}else {

}
```

Switch

Un `switch` es una declaración condicional que realiza acciones basadas en diferentes condiciones. Utiliza una comparación estricta (`===`) para hacer coincidir las condiciones y ejecuta los bloques de código de la condición coincidente. La sintaxis de la expresión `switch` se muestra a continuación.

```
switch(expression) {
  case x:
    // bloque de código
    break;
  case y:
    // bloque de código
    break;
  default:
    // bloque de código
}
```

La expresión se evalúa una vez y se compara con cada caso. Si se encuentra una coincidencia, entonces se ejecuta el bloque de código asociado y si no se encuentran coincidencias, se ejecuta el bloque de código `default` . La palabra clave `break` detiene la ejecución y se puede colocar en cualquier lugar. En su ausencia, la siguiente condición se evalúa incluso si las condiciones no coinciden.

A continuación se muestra un ejemplo de cómo obtener un nombre de día de la semana según la condición del switch.

```
switch (new Date().getDay()) {
  case 0:
    day = "Domingo";
    break;
  case 1:
    day = "Lunes";
    break;
  case 2:
     day = "Martes";
    break;
  case 3:
    day = "Miércoles";
    break;
  case 4:
    day = "Jueves";
    break;
  case 5:
    day = "Viernes";
    break;
  case 6:
    day = "Sábado";
}
```

En casos de coincidencia múltiple, se selecciona el **primer** valor coincidente; de lo contrario, se selecciona el valor predeterminado. En ausencia de un caso predeterminado y sin coincidencia, el programa continúa con las siguientes declaraciones después de las condiciones de cambio.

Exercise

A partir de los siguientes valores, escriba una sentencia `switch` que verifique el valor de diaDeLaSemana. Si `diaDeLaSemana` es "Lunes", "Martes", "Miércoles", "Jueves" o "Viernes", asigne "Es un día laborable" a la variable de resultado. Si `diaDeLaSemana` es "Sábado" o "Domingo", asigne "Es fin de semana" al resultado.

```
let diaDeLaSemana = "Lunes";
let resultado;
// comprobar si es un día laborable o el fin de semana
switch(expression) {
  case x:
    // bloque de código
    break;
  case y:
    // bloque de código
    break;
  default:
    // bloque de código
}
```

Comparadores

Centrémonos ahora en la parte condicional:

```
if (pais === "Francia") {
    ...
}
```

La parte condicional es la variable `país` seguida de los tres signos iguales (`===`). Tres signos iguales prueban si la variable `país` tiene tanto el valor correcto (`Francia`) como el tipo correcto (`String`). También puede probar condiciones con signos dobles de igual; sin embargo, un condicional como `if (x == 5)` devolvería verdadero tanto para `var x = 5;` como para `var x = "5";` . Dependiendo de lo que esté haciendo su programa, esto podría marcar una gran diferencia. Se recomienda encarecidamente como práctica recomendada que siempre compare la igualdad con tres signos iguales (`===` y `!==`) en lugar de dos (`==` y `!=`).

Otras pruebas condicionales:

- `x > a` : ¿es x mayor que a?
- `x < a` : ¿es x menor que a?
- `x <= a` : ¿es x menor o igual que a?
- `x >=a` : ¿es x mayor o igual que a?
- `x != a` : ¿es x distinto de a?
- `x` : ¿existe x?

Comparación lógica

Para evitar la molestia de si o si no, se pueden utilizar comparaciones lógicas simples.

```
let topper = marcas > 85 ? "SI" : "NO";
```

En el ejemplo anterior, `?` es un operador lógico. El código dice que si el valor de las marcas es mayor que 85, es decir, `marcas > 85` , entonces `topper = SI` ; de lo contrario `topper = NO` . Básicamente, si la condición de comparación resulta verdadera, se accede al primer argumento y si la condición de comparación es falsa, se accede al segundo argumento.

Concatenación

Además, puede concatenar diferentes condiciones con los operadores " `o` " o " `y` ", para probar si alguna de ellas es verdadera o si ambas son verdaderas, respectivamente.

En JavaScript, "o" se escribe como `||` y "y" se escribe como `&&` .

Supongamos que desea probar si el valor de x está entre 10 y 20. Podría hacerlo con una condición que indique:

```
if (x > 10 && x < 20) {
    ...
}
```

Si desea asegurarse de que el país sea "Inglaterra" o "Alemania", utilice:

```
if (pais === "Inglaterra" || pais === "Alemania") {
    ...
}
```

> **Nota**: Al igual que las operaciones con números, las condiciones se pueden agrupar usando paréntesis, por ejemplo: `if ( (nombre === "Juan" || nombre === "Genoveva") && país === "Francia")` .

Capítulo 6

Matrices

Las matrices son una parte fundamental de la programación. Una matriz es una lista de datos. Podemos almacenar una gran cantidad de datos en una variable, lo que hace que nuestro código sea más legible y más fácil de entender. También hace que sea mucho más fácil realizar funciones sobre datos relacionados.

Los datos de las matrices se denominan **elementos**.

Aquí hay una matriz simple:

```
// 1, 1, 2, 3, 5, y 8 son los elementos en esta matriz
let numeros = [1, 1, 2, 3, 5, 8];
```

Las matrices se pueden crear fácilmente usando literales de matriz o con una palabra clave `new`.

```
const coches = ["Saab", "Volvo", "BMW"]; // usando literales de matriz
const coches = new Array("Saab", "Volvo", "BMW"); // usando la palabra clave new
```

Se utiliza un número de índice para acceder a los valores de una matriz. El índice del primer elemento de una matriz es siempre `0`, ya que los índices de las matrices comienzan con `0`. El número de índice también se puede utilizar para cambiar los elementos de una matriz.

```
const coches = ["Saab", "Volvo", "BMW"];
console.log(coches[0]);
// Resultado: Saab

coches[0] = "Opel"; // cambiando el primer elemento de la matriz
console.log(coches);
// Resultado: ['Opel', 'Volvo', 'BMW']
```

> Las matrices son un tipo especial de objeto. Uno puede tener objetos en una matriz.

La propiedad `length` de una matriz devuelve el recuento de elementos numéricos. Los métodos soportados por Array se muestran a continuación:

Nombre	Descripción
`concat()`	Devuelve dos o más matrices combinadas
`join()`	Une todos los elementos de una matriz en una cadena
`push()`	Agrega uno o más elementos al final de la matriz y devuelve la longitud
`pop()`	Elimina el último elemento de una matriz y devuelve ese elemento
`shift()`	Elimina el primer elemento de una matriz y devuelve ese elemento
`unshift()`	Agrega uno o más elementos al frente de una matriz y devuelve la longitud
`slice()`	Extrae la sección de una matriz y devuelve la nueva matriz.
`at()`	Devuelve el elemento en el índice especificado o `undefined`
`splice()`	Elimina elementos de una matriz y (opcionalmente) los reemplaza y devuelve la matriz
`reverse()`	Transpone los elementos de una matriz y devuelve una referencia a una matriz
`flat()`	Devuelve una nueva matriz con todos los elementos de la submatriz concatenados en ella de forma recursiva hasta la profundidad especificada
`sort()`	Ordena los elementos de una matriz en su lugar y devuelve una referencia a la matriz
`indexOf()`	Devuelve el índice de la primera coincidencia del elemento de búsqueda.
`lastIndexOf()`	Devuelve el índice de la última coincidencia del elemento de búsqueda.
`forEach()`	Ejecuta una devolución de llamada en cada elemento de una matriz y devuelve undefined
`map()`	Devuelve una nueva matriz con un valor de retorno al ejecutar `callback` en cada elemento de la matriz.
`flatMap()`	Ejecuta `map()` seguido de `flat()` de profundidad 1
`filter()`	Devuelve una nueva matriz que contiene los elementos para los cuales `callback` devolvió `true`
`find()`	Devuelve el primer elemento para el cual `callback` devolvió `true`
`findLast()`	Devuelve el último elemento para el cual `callback` devolvió `true`
`findIndex()`	Devuelve el índice del primer elemento para el cual `callback` devolvió `true`
`findLastIndex()`	Devuelve el índice del último elemento para el cual `callback` devolvió `true`
`every()`	Devuelve `true` si `callback` devuelve `true` para cada elemento en la matriz
`some()`	Devuelve `true` si `callback` devuelve `true` para al menos un elemento en la matriz
`reduce()`	Usa `callback(acumulador, valorActual, indiceActual, matriz)` con el propósito de reducir y devuelve el valor final devuelto por la función `callback`
`reduceRight()`	Funciona de forma similar a `reduce()` pero comienza con el último elemento

Unshift

El método `unshift` agrega nuevos elementos secuencialmente al inicio o al frente de la matriz. Modifica la matriz original y devuelve la nueva longitud de la matriz. Por ejemplo.

```
let matriz = [0, 5, 10];
matriz.unshift(-5);  // 4

// RESULTADO: matriz = [-5 , 0, 5, 10];
```

> El método `unshift()` sobrescribe la matriz original.

El método `unshift` toma uno o más argumentos, los cuales representan los elementos que van a ser agregados al principio de la matriz. Agrega los elementos en el orden en que se proporcionan, por lo que el primer elemento será el primer elemento de la matriz.

A continuación se muestra un ejemplo del uso de `unshift` para agregar varios elementos a una matriz:

```
const numeros = [1, 2, 3];
const nuevaLongitud = numeros.unshift(-1, 0);
console.log(numeros); // [-1, 0, 1, 2, 3]
console.log(nuevaLongitud); // 5
```

Map

El método `Array.prototype.map()` itera sobre una matriz y modifica sus elementos usando una función callback. Luego, la función callback se aplicará a cada elemento de la matriz.

Aquí está la sintaxis para usar `map` .

```
let nuevaMatriz = viejaMatriz.map(function(elemento, indice, matriz) {
  // elemento: elemento actual que está siendo procesado en la matriz
  // indice: índice del elemento actual que está siendo procesado en la matriz
  // matriz: se invocó el mapa de matriz
  // Elemento de retorno que se agregará a nuevaMatriz
});
```

Por ejemplo, digamos que tiene una matriz de números y desea crear una nueva matriz que duplique los valores de los números en la matriz original. Podrías hacer esto usando un `map` como este.

```
const numeros = [2, 4, 6, 8];

const numerosDoblados = numeros.map(numero => numero * 2);

console.log(numerosDoblados);

// Resultado: [4, 8, 12, 16]
```

También puede usar la sintaxis de función flecha para definir la función que se pasa a `map` .

```
let doubledNumbers = numbers.map((number) => {
   return number * 2;
});
```

o

```
let doubledNumbers = numbers.map(number => number * 2);
```

El método `map()` no ejecuta la función para elementos vacíos y no cambia la matriz original.

Spread

Una matriz u objeto se puede copiar rápidamente en otra matriz u objeto utilizando el operador de extensión `(...)`. Permite expandir un iterable, como una matriz, en lugares donde se esperan cero o más argumentos (para llamadas a funciones) o elementos (para literales de matriz), o expandir una expresión de objeto en lugares donde cero o más pares clave-valor. (para literales de objetos) se esperan.

A continuación se muestran algunos ejemplos de ello:

```
let arr = [1, 2, 3, 4, 5];

console.log(...arr);
// Resultado: 1 2 3 4 5

let arr1;
arr1 = [...arr]; //copia arr a arr1

console.log(arr1);    //Resultado: [1, 2, 3, 4, 5]

arr1 = [6,7];
arr = [...arr,...arr1];

console.log(arr);   //Resultado: [1, 2, 3, 4, 5, 6, 7]
```

> El operador de propagación solo funciona en navegadores modernos que admiten esta función. Si necesita admitir navegadores más antiguos, deberá utilizar un transpilador como Babel para convertir la sintaxis del operador de extensión a código ES5 equivalente.

Shift

El método `shift` elimina el primer índice de esa matriz y mueve todos los índices hacia la izquierda. Cambia la matriz original. Aquí está la sintaxis para usar "shift":

```
array.shift();
```

Por ejemplo:

```
let array = [1, 2, 3];
array.shift();

// Resultado: array = [2,3]
```

También puede utilizar el método `shift` junto con un bucle para eliminar todos los elementos de una matriz. A continuación se muestra un ejemplo de cómo podría hacer esto:

```
while (array.length > 0) {
  array.shift();
}

console.log(array); // Resultado: []
```

> El método `shift` solo funciona en matrices, y no en otros objetos que sean similares a matrices, como objetos de argumentos u objetos NodeList. Si necesita cambiar elementos de uno de estos tipos de objetos, primero deberá convertirlo en una matriz utilizando el método `Array.prototype.slice()` .

Pop

El método `pop` elimina el último elemento de una matriz y devuelve ese elemento. Este método cambia la longitud de la matriz.

Aquí está la sintaxis para usar `pop` :

```
array.pop();
```

Por ejemplo:

```
let matriz = ["uno", "dos", "tres", "cuatro", "cinco"];
matriz.pop();

console.log(matriz);

// Resultado: ['uno', 'dos', 'tres', 'cuatro']
```

Puede también usar el método `pop` en conjunción con un bucle para eliminar todos los elementos de una matriz. Aquí tiene un ejemplo de cómo podría hacer esto:

```
while (matriz.length > 0) {
  matriz.pop();
}

console.log(matriz); // Resultado: []
```

> El método `pop` solo funciona en matrices, y no en otros objetos que sean similares a matrices, como objetos de argumentos u objetos NodeList. Si necesita extraer elementos de uno de estos tipos de objetos, primero deberá convertirlo en una matriz utilizando el método `Array.prototype.slice()` .

Join

El método `join` hace que una matriz se convierta en una cadena y lo une todo. No cambia la matriz original. Aquí está la sintaxis para usar `join`:

```
matriz.join([separador]);
```

El argumento `separador` es opcional y especifica el carácter que se usa para separar los elementos en la cadena resultante. Si se omite, los elementos de la matriz son separados con una coma (`,`).

Por ejemplo:

```
let matriz = ["uno", "dos", "tres", "cuatro"];

console.log(matriz.join(" "));

// Resultado: uno dos tres cuatro
```

> Se puede especificar cualquier separador pero el predeterminado es una coma `(,)`.

En el ejemplo anterior, se utiliza un espacio como separador. También puedes usar `join` para convertir un objeto similar a una matriz (como un objeto de argumentos o un objeto NodeList) en una cadena convirtiéndolo primero en una matriz usando el método `Array.prototype.slice()`:

```
function imprimirArgumentos() {
  console.log(Array.prototype.slice.call(argumentos).join(', '));
}

imprimirArgumentos('a', 'b', 'c'); // Resultado: "a, b, c"
```

Length

Las matrices tienen una propiedad llamada `length`, y es más o menos exactamente como suena, es la longitud de la matriz.

```
let matriz = [1, 2, 3];

let l = matriz.length;

// Resultado: l = 3
```

La propiedad length también establece el número de elementos en la matriz. Por ejemplo.

```
let frutas = ["Plátano", "Naranja", "Manzana", "Mango"];
frutas.length = 2;

console.log(frutas);
// Resultado: ['Plátano', 'Naranja']
```

También puede usar la propiedad `length` para obtener el último elemento de una matriz usándola como un índice. Por ejemplo:

```
console.log(frutas[frutas.length - 1]); // Resultado: Naranja
```

Puede también usar la propiedad `length` para agregar elementos al final de una matriz. Por ejemplo:

```
frutas[frutas.length] = "Piña";
console.log(frutas); // Resultado: ['Plátano', 'Naranja', 'Piña']
```

> La propiedad `length` se actualiza automáticamente cuando se agregan o eliminan elementos de la matriz.

También vale la pena señalar que la propiedad `length` no es un método, por lo que no es necesario utilizar paréntesis al acceder a ella. Es simplemente una propiedad del objeto Array a la que puede acceder como cualquier otra propiedad de objeto.

Push

Se pueden insertar ciertos elementos en una matriz haciendo que el último índice sea el elemento recién agregado. Cambia la longitud de una matriz y devuelve una nueva longitud.

Aquí está la sintaxis para usar `push`:

```
matriz.push(elemento1[, ...[, elementoN]]);
```

Los argumentos `elemento1, ..., elementoN` representan los elementos que serán agregados al final de la matriz.

Por ejemplo:

```
let matriz = [1, 2, 3];
matriz.push(4);

console.log(matriz);

// Resultado: matriz = [1, 2, 3, 4]
```

También puede usar `push` para agregar elementos al final de un objeto similar a una matriz (como un objeto de argumentos o un objeto NodeList) convirtiéndolo primero en una matriz usando el método `Array.prototype.slice()`:

```
function imprimirArgumentos() {
  let args = Array.prototype.slice.call(argumentos);
  args.push('d', 'e', 'f');
  console.log(args);
}

imprimirArgumentos('a', 'b', 'c'); // Resultado: ["a", "b", "c", "d", "e", "f"]
```

> **Nota**: El método `push` modifica la matriz original. No crea una nueva matriz.

For Each

El método `forEach` ejecuta una función proporcionada una vez para cada elemento de la matriz. Aquí está la sintaxis para usar `forEach`:

```
array.forEach(function(elemento, indice, matriz) {
  // elemento: elemento actual que se está procesando en la matriz
  // indice: índice del elemento actual que se está procesando en la matriz
  // matriz: la matriz donde se invoca foreach
});
```

Por ejemplo, digamos que tiene una serie de números y desea imprimir el doble de cada número en la consola. Podrías hacer esto usando `forEach` así:

```
let numeros = [1, 2, 3, 4, 5];
numeros.forEach(function(numero) {
  console.log(numero * 2);
});
```

También puedes usar la sintaxis de la función de flecha para definir la función pasada a `forEach`:

```
numeros.forEach((numero) => {
  console.log(numero * 2);
});
```

o

```
numeros.forEach(numero => console.log(numero * 2));
```

El método `forEach` no modifica la matriz original. Simplemente itera sobre los elementos de la matriz y ejecuta la función proporcionada para cada elemento.

> El método `forEach()` no se ejecuta para la declaración vacía.

Sort

El método `sort` ordena los elementos de una matriz en un orden específico (ascendente o descendente). De forma predeterminada, el método `sort` ordena los elementos como cadenas y los organiza en orden ascendente según sus valores de unidad de código UTF-16.

Aquí está la sintaxis para usar `sort` :

```
matriz.sort([funcionDeComparacion]);
```

El argumento `funcionDeComparacion` es opcional y especifica una función que define el orden de clasificación. Si se omite, los elementos se ordenan en orden ascendente según su representación en cadena.

Por ejemplo:

```
let ciudades = ["California", "Barcelona", "Paris", "Kathmandu"];
let ciudadesOrdenadas = ciudades.sort();

console.log(ciudadesOrdenadas);

// Resultado: ['Barcelona', 'California', 'Kathmandu', 'Paris']
```

> Los números se pueden ordenar incorrectamente cuando se ordenan. Por ejemplo, "35" es mayor que "100", porque "3" es mayor que "1".

Para solucionar el problema de clasificación en números, se utilizan funciones de comparación. Las funciones de comparación definen el orden de clasificación y devuelven un valor **negativo**, **cero** o **positivo** según argumentos, como este:

- Un valor negativo si `a` debe ordenarse antes que `b`
- Un valor positivo si `a` debe ordenarse después de `b`
- `0` si `a` y `b` son iguales y su orden no importa

```
const puntos = [40, 100, 1, 5, 25, 10];
puntos.sort((a, b) => {return a-b});

// Resultado: [1, 5, 10, 25, 40, 100]
```

> El método `sort()` anula la matriz original.

Índices

Entonces tienes tu conjunto de elementos de datos, pero ¿qué pasa si quieres acceder a un elemento específico? Ahí es donde entran los índices. Un **índice** se refiere a un lugar en la matriz. Los índices progresan lógicamente uno por uno, pero cabe señalar que el primer índice de una matriz es 0, como ocurre en la mayoría de los lenguajes. Los corchetes `[]` se utilizan para indicar que se está haciendo referencia a un índice de una matriz.

```
// Esto es una matriz de cadenas
let frutas = ["manzana", "plátano", "piña", "fresa"];

// Establecemos la variable platano al valor del segundo elemento de
// la matriz de frutas. Recuerde que los índices comienzan en 0, por lo que 1 es el
// segundo elemento. Resultado: plátano = "plátano"
let platano = frutas[1];
```

También puede utilizar un índice de matriz para establecer el valor de un elemento en una matriz:

```
let matriz = ['a', 'b', 'c', 'd', 'e'];
//  indices:   0    1    2    3    4
matriz[4] = 'f';
console.log(matriz); // Resultado: ['a', 'b', 'c', 'd', 'f']
```

Tenga en cuenta que si intenta acceder o configurar un elemento utilizando un índice que está fuera de los límites de la matriz (es decir, un índice menor que 0 o mayor o igual a la longitud de la matriz), obtendrá un valor `undefined`.

```
console.log(matriz[5]); // Salida: undefined
matriz[5] = 'g';
console.log(matriz); // Resultado: ['a', 'b', 'c', 'd', 'f', undefined, 'g']
```

Capítulo 7

Bucles

Los bucles son condiciones repetitivas en las que cambia una variable del bucle. Los bucles son útiles si desea ejecutar el mismo código una y otra vez, cada vez con un valor diferente.

En lugar de escribir:

```
hacerAlgo(coches[0]);
hacerAlgo(coches[1]);
hacerAlgo(coches[2]);
hacerAlgo(coches[3]);
hacerAlgo(coches[4]);
```

Puede escribir:

```
for (var i = 0; i < coches.length; i++) {
  hacerAlgo(coches[i]);
}
```

For

La forma más sencilla de bucle es la declaración for. Éste tiene una sintaxis similar a una declaración if, pero con más opciones:

```
for (condicion; condicion_finalizacion; cambio) {
    // hazlo, hazlo ahora
}
```

Veamos cómo ejecutar el mismo código diez veces usando un bucle `for` :

```
for (let i = 0; i < 10; i = i + 1) {
  // haz este código diez veces
}
```

> ***Nota***: `i = i + 1` se puede escribir `i++` .

Para recorrer las propiedades de un objeto o una matriz también se puede utilizar un bucle `for in` .

```
for (clave in objeto) {
  // bloque de código a ejecutar
}
```

A continuación se muestran ejemplos de bucle `for in` para un objeto y una matriz:

```
const persona = {nombre:"John", apellido:"Doe", edad:25};
let info = "";
for (let x in persona) {
  info += persona[x];
}

// Resultado: info = "JohnDoe25"

const numeros = [45, 4, 9, 16, 25];
let txt = "";
for (let x in numeros) {
  txt += numeros[x];
}

// Resultado: txt = '45491625'
```

El valor de objetos iterables como `Arrays` , `Strings` , `Maps` , `NodeLists` se puede iterar usando la declaración `for of` .

```
let lenguaje = "JavaScript";
let text = "";
for (let x of lenguaje) {
text += x;
}
```

While

Los bucles while ejecutan repetidamente un bloque de código siempre que una condición especificada sea verdadera. Proporciona una forma de automatizar tareas repetitivas y realizar iteraciones basadas en la evaluación de la condición.

```
while (condicion) {
  // hazlo siempre que la condición sea verdadera
}
```

Por ejemplo, el bucle en este ejemplo ejecutará repetidamente su bloque de código siempre que la variable i sea menor que 5:

```
var i = 0,
  x = "";
while (i < 5) {
  x = x + "El número es " + i;
  i++;
}
```

¡Tenga cuidado de evitar bucles infinitos si la condición siempre es verdadera!

Do...While

La instrucción do... while crea un bucle que ejecuta una instrucción específica hasta que la condición de prueba se evalúa como falsa. La condición se evalúa después de ejecutar la declaración. La sintaxis para do... while es

```
do {
  // sentencias
} while (expresión);
```

Veamos, por ejemplo, cómo imprimir números menores de 10 usando el bucle `do... while` :

```
var i = 0;
do {
  document.write(i + " ");
  i++; // incrementando i en uno
} while (i < 10);
```

> ***Nota***: `i = i + 1` se puede escribir `i++` .

Capítulo 8

Funciones

Las funciones son una de las nociones más poderosas y esenciales en programación. Las funciones como las funciones matemáticas realizan transformaciones, toman valores de entrada llamados **argumentos** y **devuelven** un valor de salida.

Las funciones se pueden crear de dos maneras: usando `declaración de función` o `expresión de función`. El *nombre de la función* se puede omitir en la `expresión de función`, convirtiéndola en una `función anónima`. Las funciones, al igual que las variables, deben declararse. Declaremos una función `doble` que acepta un *argumento* llamado `x` y **devuelve** el doble de x:

```
// un ejemplo de una declaración de función
function doble(x) {
  return 2 * x;
}
```

> *Nota:* **se puede** hacer referencia a la función anterior antes de que se haya definido.

Las funciones también son valores en JavaScript; se pueden almacenar en variables (como números, cadenas, etc.) y darse a otras funciones como argumentos:

```
// un ejemplo de una expresión de función
let double = function (x) {
  return 2 * x;
};
```

> *Nota:* **no se puede** hacer referencia a la función anterior antes de definirla, como cualquier otra variable.

Una retrollamada es una función que se pasa como argumento a otra función.

Una función de flecha es una alternativa compacta a las funciones tradicionales que tiene algunas diferencias semánticas con algunas limitaciones. Estas funciones no tienen sus propios enlaces a `this`, `arguments` y `super`, y no pueden usarse como constructores. Un ejemplo de una función de flecha:

```
const double =  (x) =>  2 * x;
```

La palabra clave `this` en la función de flecha representa el objeto que definió la función de flecha.

Funciones de orden superior

Las funciones de orden superior son funciones que manipulan otras funciones. Por ejemplo, una función puede tomar otras funciones como argumentos y/o producir una función como valor de retorno. Estas técnicas funcionales *elegantes* son construcciones poderosas disponibles en JavaScript y otros lenguajes de alto nivel como python, lisp, etc.

Ahora crearemos dos funciones simples, `add_2` y `double`, y una función de orden superior llamada `map`. `map` aceptará dos argumentos, `func` y `list` (por lo tanto, su declaración comenzará con `map(func,list)`) y devolverá una matriz. `func` (el primer argumento) será una función que se aplicará a cada uno de los elementos de la matriz `list` (el segundo argumento).

```
// Define dos funciones simples
let add_2 = function (x) {
  return x + 2;
};
let double = function (x) {
  return 2 * x;
};

// map es una función interesante que acepta 2 argumentos:
//  func    la función a llamar
//  list    una matriz de valores para llamar a la función
let map = function (func, list) {
  let output = []; // lista de salida
  for (idx in list) {
    output.push(func(list[idx]));
  }
  return output;
};

// Usamos map para aplicar una función a una lista completa
// de entradas para "asignarlas" a una lista de salidas correspondientes
map(add_2, [5, 6, 7]); // => [7, 8, 9]
map(double, [5, 6, 7]); // => [10, 12, 14]
```

Las funciones del ejemplo anterior son simples. Sin embargo, cuando se pasan como argumentos a otras funciones, se pueden componer de formas imprevistas para construir funciones más complejas.

Por ejemplo, si notamos que utilizamos las invocaciones `map(add_2,...)` y `map(double,...)` muy a menudo en nuestro código, podríamos decidir que queremos crear dos listas de propósito especial. funciones de procesamiento que tienen incorporada la operación deseada. Usando la composición de funciones, podríamos hacer esto de la siguiente manera:

```
process_add_2 = function (list) {
  return map(add_2, list);
};
process_double = function (list) {
  return map(double, list);
};
process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Ahora creemos una función llamada `buildProcessor` que toma una función `func` como entrada y devuelve un procesador `func`, es decir, una función que aplica `func` a cada entrada en la lista.

```
// una función que genera un procesador de listas que realiza
let buildProcessor = function (func) {
  let process_func = function (list) {
    return map(func, list);
  };
  return process_func;
};
// llamar a buildProcessor devuelve una función que se llama con una entrada de lista

// usando buildProcessor podríamos generar los procesadores add_2 y de lista doble de la siguiente manera:
process_add_2 = buildProcessor(add_2);
process_double = buildProcessor(double);

process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Veamos otro ejemplo. Crearemos una función llamada `buildMultiplier` que toma un número `x` como entrada y devuelve una función que multiplica su argumento por `x`:

```
let buildMultiplier = function (x) {
  return function (y) {
    return x * y;
  };
};

let double = buildMultiplier(2);
let triple = buildMultiplier(3);

double(3); // => 6
triple(3); // => 9
```

Capítulo 9

Objetos

En javascript los objetos son **mutables** porque cambiamos los valores apuntados por el objeto de referencia, en cambio, cuando cambiamos un valor primitivo estamos cambiando su referencia que ahora apunta al nuevo valor y por lo tanto los primitivos son **inmutables**. Los tipos primitivos de JavaScript son `true` , `false` , `numbers` , `strings` , `null` e `undefined` . Cualquier otro valor es un `object` . Los objetos contienen pares `propertyName` : `propertyValue` . Hay tres formas de crear un "objeto" en JavaScript:

1. literal

```
let object = {};
// Sí, ¡simplemente un par de llaves!
```

> ***Nota:*** esta es la forma **recomendada**.

2. orientada a objetos

```
let object = new Object();
```

> ***Nota:*** es casi como Java.

3. y usando `object.create`

```
let object = Object.create(proto[, propertiesObject]);
```

> ***Nota:*** crea un nuevo objeto con el objeto prototipo y las propiedades especificadas.

Propiedades

La propiedad del objeto es un par `propertyName : propertyValue`, donde **el nombre de propiedad puede ser solo una cadena**. Si no es una cadena, se convierte en una cadena. Puede especificar propiedades **al crear** un objeto **o después**. Puede haber cero o más propiedades separadas por comas.

```
let language = {
  name: "JavaScript",
  isSupportedByBrowsers: true,
  createdIn: 1995,
  author: {
    firstName: "Brendan",
    lastName: "Eich",
  },
  // ¡Sí, los objetos se pueden anidar!
  getAuthorFullName: function () {
    return this.author.firstName + " " + this.author.lastName;
  },
  // Sí, ¡las funciones también pueden ser valores!
};
```

El siguiente código demuestra cómo **obtener** el valor de una propiedad.

```
let variable = language.name;
// variable ahora contiene una cadena "JavaScript".
variable = language["name"];
// Las líneas de arriba hacen lo mismo. La diferencia es que el segundo te permite usar literalmente cualquier cadena como nom
variable = language.newProperty;
// variable ahora no está definida porque aún no hemos asignado esta propiedad.
```

El siguiente ejemplo muestra cómo **agregar** una nueva propiedad **o cambiar** una existente.

```
language.newProperty = "nuevo valor";
// Ahora el objeto tiene una nueva propiedad. Si la propiedad ya existe, se repondrá su valor.
language["newProperty"] = "valor cambiado";
// Una vez más, puede acceder a las propiedades en ambos sentidos. Se recomienda la primera (notación de puntos).
```

Mutable

La diferencia entre objetos y valores primitivos es que podemos **cambiar objetos**, mientras que los valores primitivos son **inmutables**.

Por ejemplo:

```
let miPrimitivo = "primer valor";
miPrimitivo = "otro valor";
// miPrimitivo ahora apunta a otra cadena.
let miObjeto = { clave: "primer valor" };
miObjeto.clave = "otro valor";
// miObjeto apunta al mismo objeto.
```

Puede agregar, modificar o eliminar propiedades de un objeto utilizando la notación de puntos o la notación de corchetes.

```
let objeto = {};
objeto.foo = 'bar'; // Agrega la propiedad 'foo'
objeto['baz'] = 'qux'; // Agrega la propiedad 'baz'
objeto.foo = 'quux'; // Modifica la propiedad 'foo'
delete objeto.baz; // Borra la propiedad 'baz'
```

> Los valores primitivos (como números y cadenas) son inmutables, mientras que los objetos (como matrices y objetos) son mutables.

Referencia

Los objetos **nunca se copian**. Se transmiten por referencia. Una referencia de objeto es un valor que hace referencia a un objeto. Cuando crea un objeto utilizando el operador `new` o la sintaxis literal del objeto, JavaScript crea un objeto y le asigna una referencia. A continuación se muestra un ejemplo de creación de un objeto utilizando la sintaxis literal del objeto:

```
var object = {
  foo: 'bar'
};
```

Aquí hay un ejemplo de cómo crear un objeto usando el operador `new`:

```
var object = new Object();
object.foo = 'bar';
```

Cuando asigna una referencia de objeto a una variable, la variable simplemente contiene una referencia al objeto, no al objeto en sí. Esto significa que si asigna la referencia del objeto a otra variable, ambas variables apuntarán al mismo objeto.

Por ejemplo:

```
var object1 = {
  foo: 'bar'
};

var object2 = object1;

console.log(object1 === object2); // Salida: true
```

En el ejemplo anterior, tanto `object1` como `object2` son variables que contienen referencias al mismo objeto. El operador `===` se usa para comparar las referencias, no los objetos en sí, y devuelve `true` porque ambas variables contienen referencias al mismo objeto.

> Puede utilizar el método `Object.assign()` para crear un nuevo objeto que sea una copia de un objeto existente.

A continuación se muestra un ejemplo de un objeto por referencia.

```
// Imagínate que comí una pizza
let myPizza = { slices: 5 };
// Y lo compartí contigo
let yourPizza = myPizza;
// me como otra rebanada
myPizza.slices = myPizza.slices - 1;
let numberOfSlicesLeft = yourPizza.slices;
// Ahora tenemos 4 porciones porque myPizza y yourPizza
// referencia al mismo objeto pizza.
let a = {},
  b = {},
  c = {};
// a, byc se refieren cada uno a a
// objeto vacío diferente
a = b = c = {};
// a, byc se refieren todos a
// el mismo objeto vacío
```

Prototype

Cada objeto está vinculado a un objeto prototype (en español, prototipo) del cual hereda propiedades. Los objetos creados a partir de literales de objetos (`{}`) se vinculan automáticamente a `Object.prototype` , que es un objeto que viene estándar con JavaScript.

Cuando un intérprete de JavaScript (un módulo en su navegador) intenta encontrar una propiedad que desea recuperar, como en el siguiente código:

```
let adulto = { edad: 26 },
  propiedadRecuperada = adulto.edad;
// La línea de arriba
```

Primero, el intérprete examina todas las propiedades que tiene el objeto. Por ejemplo, `adulto` solo tiene una propiedad propia: `edad` . Pero además de esa, en realidad tiene algunas propiedades más, que fueron heredadas de `Object.prototype` .

```
let cadenaPresentacion = adulto.toString();
// la variable tiene valor de '[object Object]'
```

`toString` es una propiedad de Object.prototype, que fue heredada. Tiene un valor de una función, que devuelve una representación de cadena del objeto. Si desea que devuelva una representación más significativa, puede sobreescribirla. Simplemente agregue una nueva propiedad al objeto adulto.

```
adulto.toString = function () {
  return "Tengo " + this.edad;
};
```

Si llama a la función `toString` ahora, el intérprete encontrará la nueva propiedad en el objeto y se detendrá.

De este modo, el intérprete recupera la primera propiedad que encontrará en el camino desde el objeto mismo y a través de su prototipo.

Para configurar su propio objeto como prototipo en lugar del Object.prototype predeterminado, puede invocar `Object.create` de la siguiente manera:

```
let niño = Object.create(adulto);
/* Esta forma de crear objetos nos permite reemplazar fácilmente el Object.prototype predeterminado por el que queramos. En es
niño.edad = 8;
/* Anteriormente, niño no tenía su propia propiedad edad y el intérprete tenía que buscar más allá del prototipo del objeto ni
  Ahora, cuando establezcamos la edad del niño, el intérprete no irá más lejos.
  Nota: la edad del adulto sigue siendo 26 años. */
let cadenaPresentacion = niño.toString();
// El valor es "Tengo 8".
/* Nota: no hemos anulado la propiedad toString del niño, por lo que se invocará el método del adulto. Si el adulto no tuviera
```

El prototipo del objeto `niño` es `adulto` , cuyo prototipo es `Object.prototype` . Esta secuencia de prototipos se denomina **cadena de prototipos**.

Delete

Un operador `delete` se puede usar para **eliminar una propiedad** de un objeto. Cuando se elimina una propiedad, es eliminada del objeto y no puede ser accedida o enumerada (por ejemplo, no se muestra en un bucle for-in).

Aquí está la sintaxis para usar `delete`:

```
delete object.property;
```

Por ejemplo:

```
let adulto = { edad: 26 },
  niño = Object.create(adulto);

niño.edad = 8;

delete niño.edad;

/* Elimine la propiedad de edad del niño, revelando la edad del prototipo, porque entonces no se anula.. */

let prototipoEdad = niño.edad;
// 26, porque el niño no tiene su propia propiedad de edad.
```

> El operador `delete` solo funciona con las propiedades propias de un objeto y no con las propiedades heredadas. Tampoco funciona en propiedades que tienen el atributo `configurable` establecido en `false`.

El operador `delete` no modifica la cadena de prototipos del objeto. Simplemente elimina la propiedad especificada del objeto y tampoco destruye el objeto ni sus propiedades. Simplemente hace que las propiedades sean inaccesibles. Si necesita destruir un objeto y liberar su memoria, puede establecer el objeto a `null` o usar un recolector de basura para recuperar la memoria.

Enumeración

La *enumeración* se refiere al proceso de iterar sobre las propiedades de un objeto y realizar una determinada acción para cada propiedad. Hay varias formas de enumerar las propiedades de un objeto en JavaScript.

Una forma de enumerar las propiedades de un objeto es utilizar el bucle `for-in`. El bucle `for-in` itera sobre las propiedades enumerables de un objeto en un orden arbitrario, y para cada propiedad ejecuta un bloque de código determinado.

La declaración `for in` puede recorrer todos los nombres de propiedades de un objeto. La enumeración incluirá funciones y propiedades del prototipo.

```
let fruta = {
    manzana: 2,
    naranja: 5,
    pera: 1,
  },
  oracion = "Tengo ",
  cantidad;
for (clase in fruta) {
  cantidad = fruta[clase];
  oracion += cantidad + " " + clase + (cantidad === 1 ? "" : "s") + ", ";
}
// La siguiente línea elimina la coma final.
oracion = oracion.substr(0, oracion.length - 2) + ".";
// Tengo 2 manzanas, 5 naranajas, 1 pera.
```

Otra forma de enumerar las propiedades de un objeto es utilizar el método `Object.keys()`, que devuelve una matriz de los nombres de propiedades enumerables del propio objeto.

For example:

```
let objeto = {
  foo: 'bar',
  baz: 'qux'
};

let propiedades = Object.keys(objeto);
propiedades.forEach(function(propiedad) {
  console.log(propiedad + ': ' + objeto[propiedad]);
});

// foo: bar
// baz: qux
```

Capítulo 10

Fecha y hora

El objeto `date` almacena la fecha y la hora y proporciona métodos para gestionarla. Los objetos de fecha son estáticos y utilizan la zona horaria predeterminada del navegador para mostrar la fecha como una cadena de texto completo.

Para crear `date` utilizamos un constructor `new Date()` y se puede crear de las siguientes maneras.

```
new Date()
new Date(cadena_con_una_fecha)
new Date(año,mes)
new Date(año,mes,dia)
new Date(año,mes,dia,hora)
new Date(año,mes,dia,hora,minutos)
new Date(año,mes,dia,hora,minutos,segundos)
new Date(año,mes,dia,hora,minutos,segundos,ms)
new Date(milliseconds)
```

> Los meses se pueden especificar de `0` a `11`, más de eso resultará en un desbordamiento al año siguiente.

Los métodos y propiedades admitidos por date se describen a continuación:

Nombre	Descripción
`constructor`	Devuelve la función que creó el prototipo del objeto date.
`getDate()`	Devuelve el dia (1-31) de un mes
`getDay()`	Devuelve el dia (0-6) de una semana
`getFullYear()`	Devuelve el año (4 digits)
`getHours()`	Devuelve la hour (0-23)
`getMilliseconds()`	Devuelve los milliseconds (0-999)
`getMinutes()`	Devuelve los minutos (0-59)
`getMonth()`	Devuelve el mes (0-11)
`getSeconds()`	Devuelve los segundos (0-59)
`getTime()`	Devuelve el numeric value of a specified date in milliseconds since midnight Jan 1 1970
`getTimezoneOffset()`	Devuelve el desplazamiento de la zona horaria en minutos
`getUTCDate()`	Devuelve el dia (1-31) de un mes según la hora universal
`getUTCDay()`	Devuelve el dia de la semana (0-6) según la hora universal.
`getUTCFullYear()`	Devuelve el año (4 dígitos) según la hora universal.
`getUTCHours()`	Devuelve la hora (0-23) según la hora universal.
`getUTCMilliseconds()`	Devuelve los milisegundos (0-999) según la hora universal
`getUTCMinutes()`	Devuelve los minutos (0-59) según hora universal
`getUTCMonth()`	Devuelve el mes (0-11) según la hora universal.
`getUTCSeconds()`	Devuelve los segundos (0-59) según la hora universal
`now()`	Devuelve el valor numérico en milisegundos desde la medianoche del 1 de enero de 1970
`parse()`	Analiza la cadena de fecha y devuelve el valor numérico en milisegundos desde la medianoche del 1 de enero de 1970
`prototype`	Permite agregar propiedades
`setDate()`	Establece el día de un mes
`setFullYear()`	Establece el año
`setHours()`	Establece la hora
`setMilliseconds()`	Establece los milisegundos
`setMinutes()`	Establece los minutos
`setMonth()`	Establece el mes
`setSeconds()`	Estsablece el segundo
`setTime()`	Establece la hora
`setUTCDate()`	Establece el dia del mes según la hora universal.
`setUTCFullYear()`	Establece el año según la hora universal.
`setUTCHours()`	Establece la hora según la hora universal.
`setUTCMilliseconds()`	Establece los milisegundos según la hora universal.

Nombre	Descripción
`setUTCMinutes()`	Configura los minutos según la hora universal.
`setUTCMonth()`	Establece el mes según la hora universal.
`setUTCSeconds()`	Ajusta el segundo según la hora universal.
`toDateString()`	Devuelve la fecha en formato legible por humanos.
`toISOString()`	Devuelve la fecha según el formato ISO.
`toJSON()`	Devuelve la fecha en una cadena, formateada como fecha JSON
`toLocaleDateString()`	Devuelve la fecha en una cadena usando convenciones locales
`toLocaleTimeString()`	Devuelve la hora en una cadena usando convenciones locales
`toLocaleString()`	Fecha de devolución utilizando convenciones locales
`toString()`	Devuelve una representación de cadena de la fecha especificada.
`toTimeString()`	Devuelve la parte *time* en un formato legible por humanos
`toUTCString()`	Convierte la fecha en una cadena según el formato universal.
`toUTC()`	Devuelve los milisegundos desde la medianoche del 1 de enero de 1970 en formato UTC
`valueOf()`	Devuelve el valor primitivo de `Date

Capítulo 11

JSON

Java**S**cript **O**bject **N**otation (JSON) es un formato basado en texto para almacenar y transportar datos. Los objetos Javascript se pueden convertir fácilmente a JSON y viceversa. Por ejemplo.

```
//  un objeto JavaScript
let miObj = { nombre:"Ryan", edad:30, ciudad:"Austin" };

// convertido en un JSON:
let miJSON = JSON.stringify(miObj);
console.log(miJSON);
// Resultado: '{"nombre":"Ryan","edad":30,"ciudad":"Austin"}'

//convertido de nuevo a objeto JavaScript
let JSONoriginal = JSON.parse(miJSON);
console.log(JSONoriginal);

// Resultado: {nombre: 'Ryan', edad: 30, ciudad: 'Austin'}
```

`stringify` y `parse` son los dos métodos admitidos por JSON.

Método	Descripción
`parse()`	Devuelve un objeto JavaScript de la cadena JSON analizada
`stringify()`	Devuelve una cadena JSON del objeto JavaScript

Los siguientes tipos de datos son compatibles con JSON.

- string
- number
- array
- boolean
- objeto con valores JSON válidos
- `null`

No puede ser `function` , `date` o `undefined` .

Guía de manejo de errores de JavaScript

Los errores son una parte inevitable del desarrollo de software. Manejarlos de manera efectiva es crucial para escribir código JavaScript robusto y confiable. Esta guía lo guiará a través de los fundamentos del manejo de errores en JavaScript, incluido por qué es importante, los tipos de errores y cómo usar la declaración `try...catch`.

Por qué es importante el manejo de errores

El manejo de errores es esencial por varias razones:

- **Recuperación elegante**: permite que su código se recupere sin problemas inesperados y continúe ejecutándose.
- **Experiencia de usuario**: el manejo eficaz de errores mejora la experiencia del usuario al proporcionar mensajes de error significativos.
- **Depuración**: los errores manejados correctamente facilitan la depuración, ya que puedes identificar problemas rápidamente.
- **Confiabilidad del código**: el manejo de errores garantiza que su código sea confiable y sólido, lo que reduce el riesgo de fallas.

Tipos de errores

Los errores de JavaScript se pueden clasificar en varios tipos, que incluyen:

- **Errores de sintaxis**: Errores que ocurren debido a una sintaxis incorrecta.
- **Errores de tiempo de ejecución**: errores que ocurren durante la ejecución del código.
- **Errores de lógica**: errores resultantes de una lógica defectuosa en el código.

Casos de uso comunes

Manejo de solicitudes de red que podrían fallar. Analizar y validar la entrada del usuario. Gestión de errores de bibliotecas de terceros.

Ventajas del manejo de errores

El manejo eficaz de errores ofrece varias ventajas:

- Evita la terminación del script.
- Permite el manejo controlado de errores.
- Proporciona información detallada sobre errores para la depuración.
- Mejora la confiabilidad del código y la experiencia del usuario.

Mejores prácticas

Para aprovechar al máximo el manejo de errores, considere estas mejores prácticas:

- Utilice tipos de errores específicos siempre que sea posible.
- Registrar errores para fines de depuración.
- Proporcionar mensajes de error claros y fáciles de usar.
- Manejar los errores lo más cerca posible de su origen.

Manejo de errores con try...catch

Una de las técnicas comunes de manejo de errores es el bloque try...catch, que se describe en las siguientes secciones.

- try...catch
- try...catch...finally

Conclusión

El manejo de errores es un aspecto crítico del desarrollo de JavaScript. Al comprender los tipos de errores y seguir las mejores prácticas, podrá escribir aplicaciones más confiables y fáciles de usar.

Capítulo 12

Manejo de errores

En la programación, los errores ocurren por varias razones, algunos se deben a errores de código, otros se deben a una entrada incorrecta y otras cosas imprevisibles. Cuando ocurre un error, el código se detiene y genera un mensaje de error que normalmente se ve en la consola.

try... catch

En lugar de detener la ejecución del código, podemos usar la construcción `try...catch` que permite detectar errores sin que el script muera. La construcción `try...catch` tiene dos bloques principales; `try` y luego `catch`.

```
try {
  // código...
} catch (err) {
  // manejo del error
}
```

Al principio, se ejecuta el código del bloque `try`. Si no se encuentran errores, se omite el bloque `catch`. Si ocurre un error, entonces se detiene la ejecución de `try`, moviendo la secuencia de control al bloque `catch`. La causa del error se captura en la variable `err`.

```
try {
  // código...
  alert('Bienvenido a Aprender JavaScript');
  asdk; // error la variable asdk no está definida
} catch (err) {
  console.log("Un error ha ocurrido");
}
```

`try...catch` Funciona para errores de tiempo de ejecución, lo que significa que el código debe ser ejecutable y sincrónico.

Para generar un error personalizado, se puede utilizar una declaración `throw`. El objeto de error generado por errores tiene dos propiedades principales.

- **name**: nombre del error
- **message**: detalles sobre el error

Si no necesitamos un mensaje de `error`, se puede omitir la captura.

try...catch...finally

Podemos agregar una construcción más a `try...catch` llamada `finally`, este código se ejecuta en todos los casos. es decir, después de `try` cuando no hay ningún error y después de `catch` en caso de error. La sintaxis de `try...catch...finally` se muestra a continuación.

```
try {
   // intenta ejecutar el código
} catch (err) {
    // maneja los errores
} finally {
   // se ejecuta siempre
}
```

Ejecutando código de ejemplo del mundo real.

```
try {
  alert( 'try' );
} catch (err) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

En el ejemplo anterior, el bloque `try` se ejecuta primero, seguido de `finally`, ya que no hay errores.

Exercise

Escriba una función `divideNumbers()` que tome dos argumentos, numerador y denominador, y devuelva el resultado de dividir el numerador por el denominador usando las siguientes configuraciones.

```
function divideNumbers(numerator, denominator) {
    try {
      // Pruebe la declaración para dividir el numerador por el denominador.
    } catch (error) {
      // imprimir mensaje de error
    } finally {
      // imprimir que la ejecución ha finalizado
    }
   // return result
  }
  let answer = divideNumbers(10, 2);
```

Capítulo 13

Módulos

En el mundo real, un programa crece orgánicamente para hacer frente a las necesidades de nuevas funciones. Con una base de código en crecimiento, estructurar y mantener el código requiere trabajo adicional. Aunque dará sus frutos en el futuro, es tentador descuidarlo y permitir que los programas queden profundamente enredados. En realidad, aumenta la complejidad de la aplicación, ya que uno se ve obligado a construir una comprensión holística del sistema y tiene dificultades para mirar cualquier pieza de forma aislada. En segundo lugar, hay que invertir más tiempo en desenredarlo para utilizar su funcionalidad.

Los *Módulos* vienen a evitar estos problemas. Un `módulo` especifica de qué piezas de código depende, junto con qué funcionalidad proporciona para que la utilicen otros módulos. Los módulos que dependen de otro módulo se denominan *dependencias*. Existen varias bibliotecas de módulos para organizar el código en módulos y cargarlo según demanda.

- AMD - uno de los sistemas modulares más antiguos, utilizado inicialmente por require.js.
- CommonJS - Sistema de módulos creado para el servidor Node.js.
- UMD - Sistema de módulos compatible con AMD y CommonJS.

Los módulos pueden cargarse entre sí y utilizar directivas especiales `import` y `export` para intercambiar funcionalidades y llamar funciones entre sí.

- `export` - etiqueta funciones y variables que deberían ser accesibles desde fuera del módulo actual
- `import` - importa funcionalidad desde el módulo externo

Veamos el mecanismo de `import` y `export` en módulos. Tenemos la función `sayHi` exportada desde el archivo `sayHi.js`.

```
// 📁 sayHi.js
export const sayHi = (user) => {
  alert(`Hello, ${user}!`);
}
```

La función `sayHi` se consume en el archivo `main.js` con la ayuda de la directiva `import`.

```
// 📁 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // función...
sayHi('Kelvin'); // ¡Hola, Kelvin!
```

Aquí, la directiva de importación carga el módulo importando la ruta relativa y asigna la variable `sayHi`.

Los módulos se pueden exportar de dos maneras: **Nombrado** y **Predeterminado**. Además, las exportaciones nombradas se pueden asignar en línea o individualmente.

```
// 📁 person.js

// exportaciones con nombre en línea
export const name = "Kelvin";
export const age = 30;

// de una vez
const name = "Kelvin";
const age = 30;
export {name, age};
```

Solo se puede tener una `export` predeterminada en un archivo.

```
// 🗏 message.js
const message = (name, age) => {
    return `${name} is ${age} years old.`;
};
export default message;
```

Según el tipo de exportación, podemos importarla de dos formas. Las exportaciones nombradas se construyen utilizando llaves, mientras que las exportaciones predeterminadas no.

```
import { name, age } from "./person.js"; // importación de exportación con nombre
import message from "./message.js"; // importación de exportación predeterminada
```

Al asignar módulos, debemos evitar la *dependencia circular*. La dependencia circular es una situación en la que el módulo A depende de B y B también depende de A directa o indirectamente.

Capítulo 14

Expresión regular

Una expresión regular es un objeto que puede construirse con el constructor `RegEx` o escribirse como un valor literal encerrando un patrón entre una barra diagonal `(/)`. Las sintaxis para crear una expresión regular se muestran a continuación.

```
// usando el cosntructor de expresión regular
new RegExp(patron[, banderas]);

// usando literales
/patron/modificadores
```

Las banderas son opcionales al crear una expresión regular usando literales. Un ejemplo de creación de un regular idéntico utilizando el método mencionado anteriormente es el siguiente.

```
let re1 = new RegExp("xyz");
let re2 = /xyz/;
```

Ambas formas crearán un objeto regex y tendrán los mismos métodos y propiedades. Hay casos en los que podríamos necesitar valores dinámicos para crear una expresión regular; en ese caso, los literales no funcionarán y tendrán que ir con el constructor.

> En los casos en los que queremos que una barra diagonal sea parte de una expresión regular, tenemos que escapar de la barra diagonal `(/)` con una barra invertida `(\)`.

Los diferentes modificadores que se utilizan para realizar búsquedas que no distinguen entre mayúsculas y minúsculas son:

- `g` - búsqueda global (encuentra todas las coincidencias en lugar de detenerse después de la primera)
- `i` - búsqueda que no distingue entre mayúsculas y minúsculas
- `m` - coincidencia multilínea

Los *corchetes* se utilizan en una expresión regular para encontrar un rango de caracteres. Algunos de ellos se mencionan a continuación.

- `[abc]` - Encuentra cualquier carácter entre corchetes.
- `[^abc]` - encontrar cualquier carácter, menos los que están entre corchetes
- `[0-9]` - encuentra cualquier dígito de los que están entre corchetes
- `[^0-9]` - encuantra cualquier dígito, menos los que estaán entre corchetes
- `(x|y)` - encuentre cualquiera de las alternativas separadas por |

Los *metacaracteres* son caracteres especiales que tienen un significado especial en la expresión regular. Estos caracteres se describen con más detalle a continuación:

Metacaracter	Descripción
`.`	Coincide con un solo carácter excepto nueva línea o un terminador
`\w`	Coincide con un carácter de palabra (carácter alfanumérico `[a-zA-Z0-9_]`)
`\W`	Coincide con un carácter que no es una palabra (igual que `[^a-zA-Z0-9_]`)
`\d`	Coincide con cualquier carácter de dígito (igual que `[0-9]`)
`\D`	Coincide con cualquier carácter que no sea un dígito
`\s`	Coincide con un carácter de espacio en blanco (espacios, tabulaciones, etc.)
`\S`	Coincide con un carácter que no sea un espacio en blanco
`\b`	Coincidencia al principio/final de una palabra
`\B`	Coincide pero no al principio/final de una palabra
`\0`	Coincide con un carácter `NULL`
`\n`	Coincide con un carácter de nueva línea
`\f`	Coincide con un carácter de avance de formulario
`\r`	Coincide con un carácter de retorno de carro
`\t`	Coincide con un carácter de tabulación
`\v`	Coincide un carácter de tabulación vertical
`\xxx`	Coincide con un carácter especificado por un número octal `xxx`
`\xdd`	Coincide con un carácter especificado por un número hexadecimal `dd`
`\udddd`	Coincide con un carácter especificado por un número hexadecimal `dddd`

Las propiedades y métodos admitidos por RegEx se enumeran a continuación.

Nombre	Descripción
`constructor`	Devuelve la función que creó el prototipo del objeto RegEx
`global`	Comprueba si el modificador `g` está configurado
`ignoreCase`	Comprueba si el modificador `i` está configurado
`lastIndex`	Especifica el índice en el que comenzar la próxima coincidencia.
`multiline`	Comprueba si el modificador m está configurado
`source`	Devuelve el texto de la cadena.
`exec()`	Prueba la coincidencia y devuelve la primera coincidencia; si no hay coincidencia, devuelve `null`
`test()`	Prueba la coincidencia y devuelve `true` o `false`
`toString()`	Devuelve el valor de cadena de la expresión regular.

Un método `compile()` compila la expresión regular y está en desuso.

Aquí se muestran algunos ejemplos de expresiones regulares.

```
let texto = "Las mejores cosas de la vida son gratis";
let resultado = /e/.exec(texto); // busca una coincidencia de e en una cadena
// resultado: e


let textoHolaMundo = "¡Hola mundo!";
// Busca "Hola"
let pattern1 = /Hola/g;
let result1 = pattern1.test(textoHolaMundo);
// result1: true

let pattern1String = pattern1.toString();
// pattern1String : '/Hola/g'
```

Capítulo 15

Clases

Las clases son plantillas para crear un objeto. Encapsula datos con código para trabajar con datos. La palabra clave `class` se utiliza para crear una clase. Y se utiliza un método específico llamado `constructor` para crear e inicializar un objeto creado con una `class`. A continuación se muestra un ejemplo de clase de automóvil.

```
class Automovil {
  constructor(nombre, año) {
    this.nombre = nombre;
    this.año = año;
  }
  edad() {
    let fecha = new Date();
    return fecha.getFullYear() - this.año;
  }
}

let miCoche = new Automovil("Toyota", 2021);
console.log(miCoche.edad()) // 1 (si el año actual es 2022)
```

> La clase debe definirse antes de su uso.

En el cuerpo de la clase, los métodos o constructores se definen y ejecutan en `strict mode` (modo estricto, en español). La sintaxis que no se adhiere al modo estricto genera error.

Static

La palabra clave `static` define los métodos o propiedades estáticas de una clase. Estos métodos y propiedades se llaman en la propia clase.

```
class Vehiculo {
  constructor(nombre) {
    this.nombre = nombre;
  }
  static hola(x) {
    return "Hola " + x.nombre;
  }
}
let miVehiculo = new Vehiculo("Toyota");

console.log(miVehiculo.hola()); // Esto arrojará un error
console.log(Vehiculo.hola(miVehiculo));
// Resultado: Hola Toyota
```

Se puede acceder al método estático o propiedad de otro método estático de la misma clase usando la palabra clave `this`.

Herencia

La herencia es útil para fines de reutilización del código, ya que amplía las propiedades y métodos existentes de una clase. La palabra clave `extends` se utiliza para crear una herencia de clase.

```
class Vehiculo {
  constructor(marca) {
    this.nombrecoche = marca;
  }
  presentar() {
    return 'Tengo un ' + this.nombrecoche;
  }
}

class Modelo extends Vehiculo {
  constructor(marca, mod) {
    super(marca);
    this.modelo = mod;
  }
  mostrar() {
    return this.presentar() + ', es un ' + this.modelo;
  }
}

let miCoche = new Modelo("Toyota", "Camry");
console.log(miCoche.mostrar()); // Tengo un Toyota, es un Camry
```

> El prototipo de la clase padre debe ser un `Object` o `null`.

El método `super` se usa dentro de un constructor y se refiere a la clase principal. Con esto, se puede acceder a las propiedades y métodos de la clase principal.

Modificadores de acceso

`public` , `private` , y `protected` son los tres modificadores de acceso utilizados en clase para controlar su acceso desde el exterior. De forma predeterminada, todos los miembros (propiedades, campos, métodos o funciones) son accesibles públicamente desde fuera de la clase.

```
class Coche {
  constructor(nombre) {
    this.nombre = nombre;
  }
  static hola(x) {
    return "Hola " + x.nombre;
  }
}
let miCoche = new Coche("Toyota");
console.log(Coche.hola(miCoche)); // Hola Toyota
```

Los miembros `privados` sólo pueden acceder internamente dentro de la clase y no pueden ser accesibles desde fuera. Los elementos de la clase privados deben comenzar con el carácter de la almohadilla: `#` .

```
class Coche {
  constructor(nombre) {
    this.nombre = nombre;
  }
  static hola(x) {
    return "Hola " + x.nombre;
  }
  #presenta(nombrecoche) {
    return 'Tengo un ' + this.nombrecoche;
  }
}
let miCoche = new Coche("Toyota");
console.log(miCoche.#presenta("Camry")); // Error
console.log(Coche.hola(miCoche)); // Hola Toyota
```

Solo se puede acceder a los campos `protected` (protegidos, en español) desde dentro de la clase y desde aquellos que la extienden. Estos son útiles para la interfaz interna ya que la clase heredera también obtiene acceso a la clase principal. Los elementos de la clase protegidos deben comenzar con el carácter del subrayado: `_` .

```
class Coche {
  constructor(marca) {
    this.nombrecoche = marca;
  }
  _presenta() {
    return 'Tengo un ' + this.nombrecoche;
  }
}

class Modelo extends Coche {
  constructor(marca, modelo) {
    super(marca);
    this.modelo = modelo;
  }
  muestra() {
    return this._presenta() + ', es un ' + this.modelo;
  }
}
let miCoche = new Modelo("Toyota", "Camry");
```

Capítulo 16

Modelo de Objetos del Navegador (BOM en inglés)

El modelo de objetos del navegador nos permite interactuar con la ventana del navegador. El objeto `window` representa la ventana del navegador y es compatible con todos los navegadores.

El objeto `window` es el objeto predeterminado para un navegador, por lo que podemos especificar `window` o llamar directamente a todas las funciones.

```
window.alert("Bienvenido a Aprender JavaScript");

alert("Bienvenido a Aprender JavaScript")
```

De manera similar, podemos llamar a otras propiedades debajo del objeto `window` (ventana), como `history` (historial), `screen` (pantalla), `navigator` (navegador), `location` (ubicación), etc.

Window

El objeto `window` representa la ventana del navegador y es compatible con los navegadores. Las variables, objetos y funciones globales también forman parte del objeto de ventana.

Global **variables** are **properties** and **functions** are **methods** of the window object.

Las **variables** globales son **propiedades** y las **funciones** son **métodos** del objeto window.

Let's take an example of the screen properties. It is used to determine the size of the browser window and is measured in pixels.

Tomemos un ejemplo de las propiedades de la pantalla. Se utiliza para determinar el tamaño de la ventana del navegador y se mide en píxeles.

- `window.innerHeight` - la altura interior de la ventana del navegador
- `window.innerWidth` - el ancho interior de la ventana del navegador

> ***Nota***: `window.document` es lo mismo que `document.location` ya que el modelo de objeto de documento(DOM) es parte del objeto de ventana.

Algunos ejemplos de los métodos de ventana.

- `window.open()` - abre una nueva ventana
- `window.close()` - cierra la ventana actual
- `window.moveTo()` - mueve la ventana actual
- `window.resizeTo()` - cambia el tamaño de la ventana actual

Ventanas emergentes

Las ventanas emergentes son una forma adicional de mostrar información, recibir confirmación del usuario o recibir información del usuario de documentos adicionales. Una ventana emergente puede navegar a una nueva URL y enviar información a la ventana de apertura. **Cuadro de alerta** (función `alert()`), **Cuadro de confirmación** (función `confirm()`) y **Cuadro de mensaje** (función `prompt()`) son las funciones globales donde podemos mostrar la información emergente.

1. **alert()**: Muestra información al usuario y tiene un botón "**OK**" para continuar.

```
alert("Ejemplo de mensaje de alerta");
```

2. **confirm()**: Utilícelo como cuadro de diálogo para confirmar o aceptar algo. Tiene los botones "**Aceptar**" y "**Cancelar**" para continuar. Si el usuario hace clic en "**Aceptar**", devuelve `true` , si hace clic en "**Cancelar**", devuelve `false` .

```
let txt;
if (confirm("¡Presiona un botón!")) {
  txt = "¡Presionó Aceptar!";
} else {
  txt = "¡Presionó Cancelar!";
}
```

3. **prompt()**: Toma el valor ingresado por el usuario con los botones "**Aceptar"** y "**Cancelar"**. Devuelve "nulo" si el usuario no proporciona ningún valor de entrada.

```
//sintaxis
//window.prompt("alguntexto","textopredeterminado");

let persona = prompt("Por favor, introduzca su nombre", "Harry Potter");

if (person == null || person == "") {
  txt = "El usuario canceló el mensaje.";
} else {
  txt = "¡Hola, " + person + "! ¿Cómo estás hoy?";
}
```

Screen

El objeto `screen` contiene la información sobre la pantalla en la que se está representando la ventana actual. Para acceder al objeto `screen` podemos usar la propiedad `screen` del objeto `window`.

```
window.screen
//o
screen
```

El objeto `window.screen` tiene diferentes propiedades, algunas de ellas se enumeran aquí:

Propiedad	Descripción
`height`	Representa la altura de píxeles de la pantalla.
`left`	Representa la distancia en píxeles del lado izquierdo de la pantalla actual.
`pixelDepth`	Una propiedad de solo lectura que devuelve la profundidad de bits de la pantalla.
`top`	Representa la distancia en píxeles de la parte superior de la pantalla actual.
`width`	Representa el ancho de píxeles de la pantalla.
`orientation`	Devuelve la orientación de la pantalla como se especifica en la IPA de Orientación de Pantalla.
`availTop`	Una propiedad de solo lectura que devuelve el primer píxel desde la parte superior que no está ocupado por los elementos del sistema.
`availWidth`	Una propiedad de solo lectura que devuelve el ancho de píxeles de la pantalla excluyendo los elementos del sistema.
`colorDepth`	Una propiedad de solo lectura que devuelve el número de bits utilizados para representar los colores.

Navigator

`window.navigator` o `navigator` es una propiedad **de solo lectura** y contiene diferentes métodos y funciones relacionadas con el navegador.

Veamos algunos ejemplos de de uso.

1. **navigator.appName**: Da el nombre de la aplicación del navegador.

```
navigator.appName;
// "Netscape"
```

> ***Nota:*** "Netscape" es el nombre de la aplicación para IE11, Chrome, Firefox y Safari.

2. **navigator.cookieEnabled**: Devuelve un valor booleano basado en el valor de la cookie en el navegador.

```
navigator.cookieEnabled;
//true
```

3. **navigator.platform**: Proporciona información sobre el sistema operativo del navegador.

```
navigator.platform;
"MacIntel"
```

Cookies 🍪

Las cookies son piezas de información que se almacenan en una computadora y a las que puede acceder el navegador.

La comunicación entre un navegador web y el servidor no tiene estado, lo que significa que trata cada solicitud de forma independiente. Hay casos en los que necesitamos almacenar información del usuario y ponerla a disposición del navegador. Con las cookies, se puede recuperar información del ordenador cuando sea necesario.

Las cookies se guardan en un par nombre-valor.

```
book = Learn Javascript
```

La propiedad `document.cookie` se utiliza para crear, leer y eliminar cookies. Crear una cookie es bastante fácil; debe proporcionar el nombre y el valor.

```
document.cookie = "book=Learn Javacript";
```

De forma predeterminada, una cookie se elimina cuando se cierra el navegador. Para que sea persistente, debemos especificar la fecha de vencimiento (en hora UTC).

```
document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC";
```

Podemos agregar un parámetro para saber a qué ruta pertenece la cookie. Por defecto, la cookie pertenece a la página actual.

```
document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
```

A continuación se muestra un ejemplo sencillo de una cookie.

```
let cookies = document.cookie;
// una forma sencilla de recuperar todas las cookies.

document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
// configura una cookie
```

History

Cuando abrimos un navegador web y navegamos por una página web, se crea una nueva entrada en la pila del historial. A medida que seguimos navegando por diferentes páginas, se introducen nuevas entradas en la pila.

Para acceder al objeto de historial podemos usar

```
window.history
// o
history
```

Para navegar entre las diferentes pilas de historial podemos usar los métodos `go()` , `forward()` y `back()` del objeto **history**.

1. **go()**: Se utiliza para navegar por la URL específica de la pila del historial.

   ```
   history.go(-1); // mueve la página hacia atrás
   history.go(0);  // actualiza la página actual
   history.go(); // actualiza la página actual
   history.go(1) // mueve la página hacia adelante
   ```

   > ***Nota:*** la posición actual de la página en la pila del historial es **0**.

2. **back()**: Para navegar por la página hacia atrás utilizamos el método `back()` .

   ```
   history.back();
   ```

3. **forward()**: Carga la siguiente lista en el historial del navegador. Es similar a hacer clic en el botón de avance en el navegador.

   ```
   history.forward();
   ```

Location

El objeto `location` se utiliza para recuperar la ubicación actual (URL) del documento y proporciona diferentes métodos para manipular la ubicación del documento. Se puede acceder a la ubicación actual mediante

```
window.location
//o
document.location
//o
location
```

> ***Nota***: `window.location` y `document.location` hacen referencia al mismo objeto `location`.

Tomemos un ejemplo de la siguiente URL y exploremos las diferentes propiedades de `location`

`http://localhost:3000/js/index.html?type=listing&page=2#title`

```
location.href //imprime la URL del documento actual
location.protocol //imprime protocolo como http: o https:
location.host //imprime el nombre de host con un puerto como localhost o localhost:3000
location.hostname //imprime el nombre de host como localhost o www.example.com
location.port //imprime el número de puerto como 3000
location.pathname //imprime el nombre de la ruta como /js/index.html
location.search //imprime una cadena de consulta como ?type=listing&page=2
location.hash //imprime el identificador del fragmento como #title
```

Capítulo 17

Eventos

En programación, los *eventos* son acciones o sucesos en un sistema sobre los que el sistema le informa para que pueda responder a ellos. Por ejemplo, cuando hace clic en el botón Restablecer, se borra la entrada.

Las interacciones del teclado, como las pulsaciones de teclas, deben leerse constantemente para captar el estado de la tecla antes de volver a soltarla. Realizar otros cálculos que requieren mucho tiempo puede hacer que no presione una tecla. Este solía ser el mecanismo de manejo de entradas de algunas máquinas primitivas. Un paso más adelante es utilizar una cola, es decir, un programa que comprueba periódicamente la cola en busca de nuevos eventos y reacciona ante ellos. Este enfoque se llama *pooling*.

El principal inconveniente de este enfoque es que tiene que mirar la cola de vez en cuando, lo que provoca interrupciones cuando se activa un evento. El mejor mecanismo para esto es notificar al código cuando ocurre un evento. Esto es lo que hacen los navegadores modernos al permitirnos registrar funciones como *controladores* para eventos específicos.

```
<p>Haz clic en mí para activar el controlador.</p>
<script>
  window.addEventListener("click", () => {
    console.log("hizo clic");
  });
</script>
```

Aquí, se llama a `addEventListener` en el objeto `window` (objeto integrado proporcionado por el navegador) para registrar un controlador para toda la `window`. Llamar a su método `addEventListener` registra el segundo argumento que se llamará cada vez que ocurra el evento descrito por su primer argumento.

> Los detectores de eventos se llaman solo cuando el evento ocurre en el contexto del objeto en el que están registrados.

Algunos de los eventos HTML comunes se mencionan aquí.

Evento	Descripción
`onchange`	Cuando el usuario cambia o modifica el valor de la entrada del formulario
`onclick`	Cuando el usuario hace clic en el elemento.
`onmouseover`	Cuando el cursor del mouse pasa sobre el elemento
`onmouseout`	Cuando el cursor del mouse sale del elemento.
`onkeydown`	Cuando el usuario presiona y luego suelta la tecla
`onload`	Cuando el navegador haya terminado de cargar

Es común que los controladores registrados en nodos con hijos también reciban eventos de los hijos. Por ejemplo, si se hace clic en un botón dentro de un párrafo, los controladores registrados en el párrafo también recibirán el evento de clic. En caso de presencia de manejadores en ambos, el de abajo irá primero. Se dice que el evento se propaga hacia afuera, desde el nodo iniciador hasta su nodo padre y en la raíz del documento.

El controlador de eventos puede llamar al método `stopPropagation` en el objeto del evento para evitar que los controladores más arriba reciban el evento. Esto es útil en casos como, tienes un botón dentro de un elemento en el que se puede hacer clic y no deseas activar el comportamiento de clic del elemento externo al hacer clic en un botón.

```
<p>Un párrafo con un <button>botón</button>.</p>
<script>
  let parrafo = document.querySelector("p"),
  boton = document.querySelector("button");
  parrafo.addEventListener("mousedown", () => {
    console.log("Manejador de párrafos.");
  });
  boton.addEventListener("mousedown", event => {
    console.log("Manejador de boton.");
    event.stopPropagation();
  });
</script>
```

Aquí, los controladores *`mousedown`* se registran tanto por párrafo como por botón. Al hacer clic en el botón, el controlador del botón llama a "stopPropagation", lo que evitará que se ejecute el controlador del párrafo.

Los eventos pueden tener un comportamiento predeterminado. Por ejemplo, los enlaces navegan hasta el destino del enlace al hacer clic, se navega al final de una página al hacer clic en la flecha hacia abajo, y así sucesivamente. Estos comportamientos predeterminados se pueden evitar llamando a un método `preventDefault` en el objeto del evento.

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("No.");
    event.preventDefault();
  });
</script>
```

Aquí, se evita el comportamiento predeterminado del enlace al hacer clic, es decir, navegar hacia el destino del enlace.target.

Capítulo 18

Promise, async/await (Promesas y asincronía)

Las promesas (los objetos Promise) ahora son utilizadas por la mayoría de las API modernas. Por tanto, es importante comprender cómo funcionan y saber cómo utilizarlos para optimizar su código. En este capítulo, definiremos en detalle qué son las promesas y cómo usarlas en operaciones asincrónicas.

Capítulo 18

Promise, async/await

Imagine que es un escritor de libros popular y planea publicar un nuevo libro en un día determinado. Los lectores que estén interesados en este libro lo agregarán a su lista de deseos y recibirán una notificación cuando se publique o incluso si el día de lanzamiento también se pospuso. El día del lanzamiento, todos reciben una notificación y pueden comprar el libro, haciendo felices a todas las partes. Esta es una analogía de la vida real que ocurre en la programación.

1. Un "*código productor*" es algo que lleva tiempo y logra algo. Aquí es un escritor de libros.
2. Un "*código consumidor*" es alguien que consume el "código productor" una vez que está listo. En este caso, es un "lector".
3. El vínculo entre el "*código productor*" y el "*código consumidor*" puede denominarse *promesa* ya que garantiza la obtención de los resultados del "*código productor*" al "*código consumidor*".

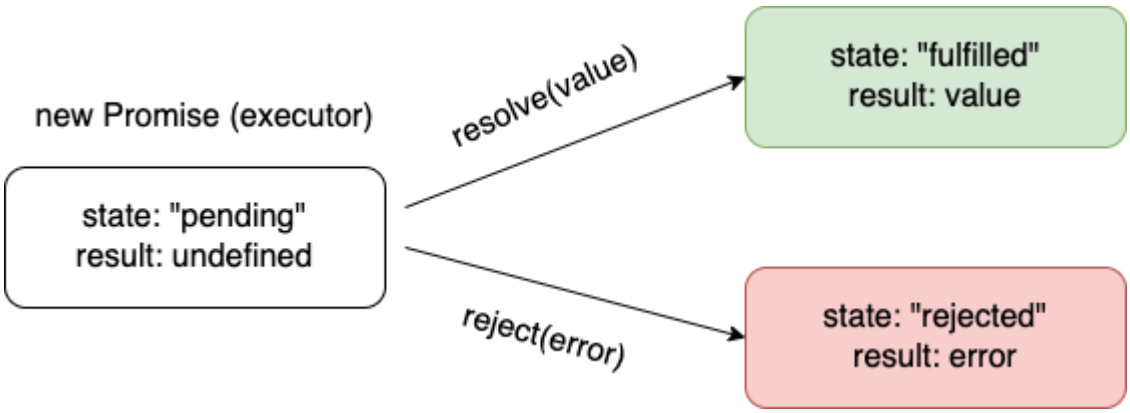
Promise

La analogía que hicimos también es válida para el objeto `promise` de JavaScript. La sintaxis del constructor para el objeto `promise` es:

```
let promesa = new Promise(function(resuelve, rechaza) {
  // Ejecutor (el código productor, "escritor")
});
```

Aquí, se pasa una función a `new Promise`, también conocida como *ejecutor*, y se ejecuta automáticamente al momento de su creación. Contiene el código productor que da el resultado. `resuelve` y `rechaza` son los argumentos proporcionados por el propio JavaScript y se llama a uno de ellos cuando el código productos da los resultados.

- `resuelve(valor)` : una retrollamada que devuelve "valor" como resultado
- `rechaza(error)` : una retrollamada que devuelve `error` en caso de error y devuelve un objeto de error como resultado



Las propiedades internas del objeto `promise` devuelta por el constructor `new Promise` son las siguientes:

- `state` - inicialmente `pending`, luego cambia a `fulfill` al resolverse o `rejected` cuando se llama a `reject`
- `result` - inicialmente `undefined`, luego cambia a `value` al resolverse `resolve` o `error` cuando se llama a `reject`

> No se puede acceder a las propiedades de la promesa: `state` y `result`. Se necesitan métodos de promesa para manejar las promesas.

Ejemplo de una promesa.

```
let promesaUno = new Promise(function(resuelve, rechaza) {
  // la función se ejecuta automáticamente cuando se construye la promesa

  // después de 1 segundo da señal de que el trabajo ha terminado con el resultado "hecho"
  setTimeout(() => resuelve("hecho"), 1000);
});

let promesaDos = new Promise(function(resuelve, rechaza) {
  // la función se ejecuta automáticamente cuando se construye la promesa

  // después de 1 segundo da señal de que el trabajo ha terminado con el resultado "error"
  setTimeout(() => rechaza(new Error("¡Vaya!")), 1000);
});
```

Aquí, `promesaUno` es un ejemplo de una "*promesa cumplida*" ya que resuelve exitosamente los valores, mientras que `promesaDos` es una "*promesa rechazada*" ya que es rechazada. Una promesa que se rechaza o se resuelve se denomina promesa *resuelta*, a diferencia de una promesa inicialmente *pendiente*. La función de consumo de la promesa se puede registrar utilizando los métodos `.then` y `.catch`. También podemos agregar el método `.finally` para realizar la limpieza o finalizar después de que se hayan completado los métodos anteriores.

```
let promesaUno = new Promise(function(resuelve, rechaza) {
  setTimeout(() => resuelve("¡hecho!"), 1000);
});

// resuelve ejecuta la primera función en .then
promesaUno.then(
  resuelve => alert(resuelve), // muestra "¡hecho! después de un segundo
  error => alert(error) // no se ejecuta
);

let promesaDos = new Promise(function(resuelve, rechaza) {
  setTimeout(() => rechaza(new Error("¡Vaya!")), 1000);
});

// rechaza ejecuta la segunda función en .then
promesaDos.then(
  resuelve => alert(resuelve), // no se ejecuta
  error => alert(error) // muestra "Error: ¡Vaya!" después de un segundo
);

let promesaTres = new Promise((resuelve, rechaza) => {
  setTimeout(() => rechaza(new Error("¡Vaya!")), 1000);
});

// .catch(f) es lo mismo que promise.then(null, f)
promesaTres.catch(alert); // muestra "Error: ¡vaya!" después de un segundo
```

En el método `Promise.then()`, ambas retrollamadas son opcionales.

Async/Await

Con las promesas, se puede usar una palabra clave `async` para declarar una función asincrónica que devuelve una promesa, mientras que la sintaxis `await` hace que JavaScript espere hasta que esa promesa se establezca y devuelva su valor. Estas palabras clave facilitan la redacción de promesas. A continuación se muestra un ejemplo de `async` .

```
//función asíncrona f
async function f() {
  return 1;
}
// promesa que está siendo resuelta
f().then(alert); // 1
```

El ejemplo anterior se puede escribir de la siguiente manera:

```
function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

`async` garantiza que la función devuelva una promesa y envuelve las no promesas en ella. Con `await` , podemos hacer que JavaScript espere hasta que la promesa se resuelva con su valor devuelto.

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("¡Bienvenido a Aprender JavaScript!"), 1000)
  });

  let result = await promise; // espere hasta que la promesa se resuelva (*)
  alert(result); // "¡Bienvenido a Aprender JavaScript!"
}

f();
```

La palabra clave `await` solo se puede usar dentro de una función `async` .

Capítulo 19

Misceláneas

En este capítulo discutiremos varios temas que surgirán al escribir código. Los temas se enumeran a continuación:

- Literales de plantilla
- Hoisting (Alzado)
- Currying
- Polyfills y transpiladores
- Lista enlazada
- Huella global
- Depuración
- Retrollamadas
- IPA Web y AJAX
- Naturaleza de un solo hilo
- ECMAScript
- Creación e implementación de aplicaciones JS
- Pruebas

Literales de plantilla

Los literales de plantilla son literales delimitados con comillas invertidas (``) y se usan en la interpolación de variables y expresiones en cadenas.

```
let texto1 = `¡Hola mundo!`;
// literales de plantilla con código simple y doble dentro de una sola cadena
let texto2 = `A menudo lo llaman "Johnny"`;
// literales de plantilla con cadenas multilínea
let texto3 =
`El rápido
zorro marrón
salta
sobre el perro perezoso`;

// literales de plantilla con interpolación variable
const nombre = "John";
const apellido = "Doe";

const textoBienvenida = `¡Bienvenido ${nombre}, ${apellido}!`;

// plantilla de literales con interpolación de expresiones
const precio = 10;
const IVA = 0.25;

const total = `Total: ${(precio * (1 + IVA)).toFixed(2)}`;
```

Hoisting (Alzado)

El alojamiento es un comportamiento predeterminado en JavaScript para mover declaraciones en la parte superior. Mientras se ejecuta un código, se crea un contexto de ejecución global: creación y ejecución. En la fase de creación, JavaScript mueve la declaración de variables y funciones a la parte superior de la página, lo que se conoce como elevación.

```
// elevación de la variable
console.log(contador);
let contador = 1; // lanza la excepción ReferenceError: contador is not defined
```

Aunque el `contador` está presente en la memoria del montón pero no se ha inicializado, genera un error. Esto sucede debido al izado (hoisting, en inglés), la variable `contador` se iza aquí.

```
    // función de elevación
    const x = 20,
    y = 10;

    let resultado = agregar(x,y); // ❌ Uncaught ReferenceError: add is not defined
    console.log(resultado);

    let agregar = (x, y) => x + y;
```

Aquí, la función `agregar` se activa e inicializa con `undefined` en la memoria del montón en la fase de creación del contexto de ejecución global. Por lo tanto, arroja un error.

Currying

`Currying` es una técnica avanzada en programación funcional para transformar una función con múltiples argumentos en una secuencia de funciones anidadas. Transforma una función invocable como `f(a,b,c)` a invocable como `f(a)(b)(c)`. No llama a una función, sino que la transforma.

Para comprender mejor la técnica del currying, creemos una función simple `agregar` que toma tres argumentos y devuelve la suma de ellos. Luego, creamos una función `agregarCurry` que toma una única entrada y devuelve una serie de funciones con su suma.

```
// Versión normal
const agregar = (a, b, c)=>{
    return a+ b + c
}
console.log(agregar(2, 3, 5)) // 10

// Versión cambiada con la técnica de currying
const agregarCurry = (a) => {
    return (b)=>{
        return (c)=>{
            return a + b + c
        }
    }
}
console.log(agregarCurry(2)(3)(5)) // 10
```

Aquí podemos ver que tanto la versión con curry como la sin curry arrojaron el mismo resultado. El curry puede ser beneficioso por muchas razones, algunas de las cuales se mencionan aquí.

- Ayuda a evitar pasar la misma variable una y otra vez.
- Divide la función en partes más pequeñas con una única responsabilidad, lo que hace que la función sea menos propensa a errores.
- Se utiliza en programación funcional para crear una función de alto orden.

Polyfills y transpiladores

JavaScript evoluciona de vez en cuando. Regularmente, se envían, analizan y agregan nuevas propuestas de lenguaje a https://tc39.github.io/ecma262/ y luego se incorporan a la especificación. Puede haber diferencias en cómo se implementa en los motores JavaScript según el navegador. Algunos pueden implementar las propuestas preliminares, mientras que otros esperan hasta que se publique la especificación completa. Surgen problemas de compatibilidad con versiones anteriores a medida que se introducen cosas nuevas.

Para admitir el código moderno en navegadores antiguos utilizamos dos herramientas: `transpiladores` y `polyfills`.

Transpiladores

Es un programa que traduce código moderno y lo reescribe utilizando estructuras de sintaxis más antiguas para que el motor más antiguo pueda entenderlo. Por ejemplo, el operador coalescente "nulo" `??` se introdujo en 2020 y los navegadores obsoletos no pueden entenderlo.

Ahora, el trabajo del transpilador es hacer que el operador coalescente "nulo" `??` sea comprensible para los navegadores antiguos.

```
// antes de ejecutar el transpilador
height = height ?? 200;

// después de ejecutar el transpilador
height = (height !== undefined && height !== null) ? height: 200;
```

Babel es uno de los transpiladores más destacados. En el proceso de desarrollo, podemos utilizar herramientas de compilación como webpack o parcel para transpilar código.

Polyfills

Hay ocasiones en las que las nuevas funciones no están disponibles en motores de navegador obsoletos. En este caso, el código que utiliza la nueva funcionalidad no funcionará. Para llenar los vacíos, agregamos la funcionalidad que falta que se llama `polyfill`. Por ejemplo, el método `filter()` se introdujo en ES5 y no es compatible con algunos de los navegadores antiguos. Este método acepta una función y devuelve una matriz que contiene solo los valores de la matriz original para los cuales la función devuelve `true`.

```
const arr = [1, 2, 3, 4, 5, 6];
const filtered = arr.filter((e) => e % 2 === 0); // filtra el número par
console.log(filtered);

// [2, 4, 6]
```

El polyfill para el filtro es.

```
Array.prototype.filter = function (retrollamada) {
  // Almacena la nueva matriz
  const resultado = [];
  for (let i = 0; i < this.length; i++) {
    // llama a la retrollamada con el elemento, índice y contexto actuales.
     //si pasa el texto, agregue el elemento en la nueva matriz.
    if (retrollamada(this[i], i, this)) {
      resultado.push(this[i]);
    }
  }
  //return the array
  return resultado
}
```

caniuse muestra la funcionalidad y la sintaxis actualizadas admitidas por diferentes motores de navegador.

Lista Enlazada

Es una estructura de datos común que se encuentra en todos los lenguajes de programación. Una lista enlazada es muy similar a una matriz normal en Javascript, solo que actúa un poco diferente.

Aquí cada elemento de la lista es un objeto independiente que contiene un enlace o un puntero al siguiente. No existe un método o función integrado para listas vinculadas en Javascript, por lo que hay que implementarlo. A continuación se muestra un ejemplo de una lista enlazada.

```
["one", "two", "three", "four"]
```

Tipos de listas enlazadas

Hay tres tipos diferentes de listas enlazadas:

1. **Listas enlazadas individualmente:** Cada nodo contiene solo un puntero al siguiente nodo.
2. **Listas doblemente enlazadas:** Hay dos punteros en cada nodo, uno al siguiente nodo y otro al nodo anterior.
3. **Listas enlazadas circulares:** Una lista enlazada circular forma un bucle al hacer que el último nodo apunte al primer nodo o a cualquier otro nodo anterior.

agregar (add)

El método `agregar` se crea aquí para agregar valor a una lista vinculada.

```
class Nodo {
    constructor(datos) {
        this.datos = datos
        this.siguiente = null
    }
}

class ListaEnlazada {
    constructor(cabeza) {
        this.cabeza = cabeza
    }
    agregar = (valor) => {
        const nuevoNodo = new Nodo(valor)
        let actual = this.cabeza
        if (!this.cabeza) {
            this.cabeza = nuevoNodo
            return
        }
        while (actual.siguiente) {
            actual = actual.siguiente
        }
        actual.siguiente = nuevoNodo
    }
}
```

extraer (pop)

Aquí, se crea un método `extraer` para eliminar un valor de la lista vinculada.

```
class Nodo {
    constructor(datos) {
        this.datos = datos
        this.siguiente = null
    }
}

class ListaEnlazada {
    constructor(cabeza) {
        this.cabeza = cabeza
    }
    extraer = () => {
        let actual = this.cabeza
        while (actual.siguiente.siguiente) {
            actual = actual.siguiente
        }
        actual.siguiente = actual.siguiente.siguiente
    }
}
```

anteponer (prepend)

Aquí, se crea un método `anteponer` para agregar un valor antes del primer hijo de la lista vinculada.

```
class Nodo {
    constructor(datos) {
        this.datos = datos
        this.siguiente = null
    }
}

class ListaEnlazada {
    constructor(cabeza) {
        this.cabeza = cabeza
    }
    anteponer = (valor) => {
        const nuevoNodo = new Nodo(valor)
        if (!this.cabeza) {
            this.cabeza = nuevoNodo
        }
        else {
            nuevoNodo.siguiente = this.cabeza
            this.cabeza = nuevoNodo
        }
    }
}
```

eliminarPrimero (shift)

Aquí, se crea el método `eliminarPrimero` para eliminar el primer elemento de la Lista Enlazada.

```
class Nodo {
    constructor(datos) {
        this.datos = datos
        this.siguiente = null
    }
}

class ListaEnlazada {
    constructor(cabeza) {
        this.cabeza = cabeza
    }
    eliminarPrimero = () => {
        this.cabeza = this.cabeza.siguiente
    }
}
```

Huella global

Si está desarrollando un módulo, que podría estar ejecutándose en una página web, que también ejecuta otros módulos, debe tener cuidado con la superposición de nombres de variables.

Supongamos que estamos desarrollando un módulo contador:

```
let myCounter = {
  number: 0,
  plusPlus: function () {
    this.number = this.number + 1;
  },
  isGreaterThanTen: function () {
    return this.number > 10;
  },
};
```

> ***Nota:*** Esta técnica se usa a menudo con los cierres (closures, en inglés), para hacer que el estado interno sea inmutable desde el exterior.

El módulo ahora toma solo un nombre de variable: `myCounter`. Si cualquier otro módulo de la página utiliza nombres como `number` o `isGreaterThanTen`, entonces es perfectamente seguro porque no sobreescribiremos los valores de los demás.

Depuración

En programación, pueden ocurrir errores al escribir código. Podría deberse a errores sintácticos o lógicos. El proceso de encontrar errores puede llevar mucho tiempo y ser complicado y se denomina depuración de código.

Afortunadamente, la mayoría de los navegadores modernos vienen con depuradores integrados. Estos depuradores se pueden activar y desactivar, lo que obliga a informar los errores. También es posible configurar puntos de interrupción durante la ejecución del código para detener la ejecución y examinar las variables. Para esto, hay que abrir una ventana de depuración y colocar la palabra clave `debugger` en el código JavaScript. La ejecución del código se detiene en cada punto de interrupción, lo que permite a los desarrolladores examinar los valores de JavaScript y reanudar la ejecución del código.

También se puede utilizar el método `console.log()` para imprimir los valores de JavaScript en la ventana del depurador.

```
const a = 5, b = 6;
const c = a + b;
console.log(c);
// Resultado : c = 11;
```

Herramientas de desarrollo del navegador

Los navegadores modernos vienen equipados con potentes herramientas de desarrollo que ayudan a depurar JavaScript, inspeccionar HTML y CSS y monitorear las solicitudes de red. A continuación se ofrece una breve descripción general de algunas herramientas esenciales:

Chrome DevTools: Las herramientas para desarrolladores de Google Chrome ofrecen una amplia gama de funciones para depurar aplicaciones web.

Firefox DevTools: Las herramientas de desarrollo de Mozilla Firefox son otra excelente opción que ofrece capacidades similares.

Microsoft Edge DevTools: Para los usuarios de Microsoft Edge, las herramientas de desarrollo integradas brindan funciones de depuración esenciales.

Safari Web Inspector: Web Inspector de Safari es un sólido conjunto de herramientas para depurar y crear perfiles de aplicaciones web.

Usando puntos de interrupción

Los navegadores modernos ofrecen herramientas para desarrolladores con capacidades de depuración. Establezca puntos de interrupción para pausar la ejecución del código e inspeccionar variables y pilas de llamadas. Recorra el código para comprender su flujo. Herramientas de desarrollo del navegador

Los navegadores proporcionan un conjunto de herramientas para desarrolladores que le permiten inspeccionar HTML, CSS y JavaScript. Puede acceder a ellos haciendo clic derecho en una página web y seleccionando "Inspeccionar" o presionando `F12` o `Ctrl+Shift+I` . Las características clave incluyen:

Consola: Ver e interactuar con la salida de la consola.

Elementos: Inspeccionar y modificar el DOM.

Fuentes: Depurar JavaScript con puntos de interrupción y observar expresiones.

Red: Supervise las solicitudes y respuestas de la red.

Usando la declaración del depurador

Inserte la declaración `debugger` en su código para crear puntos de interrupción mediante programación. Cuando el código encuentre el depurador, pausará la ejecución y abrirá las herramientas de desarrollo del navegador.

Funciones de retrollamada en JavaScript

Las funciones de retrollamada son un concepto fundamental en JavaScript, que permite la programación asincrónica y basada en eventos. Este documento de Markdown proporciona una explicación detallada de las funciones de retrollamada, su propósito y cómo usarlas de manera efectiva.

¿Qué es una función de retrollamada?

- Una **función de retrollamada** es una función de JavaScript que se pasa como argumento a otra función.
- Por lo general, se invoca o ejecuta más adelante, a menudo después de alguna operación o evento asincrónico.
- Las retrollamadas son esenciales para manejar tareas como la recuperación de datos, el manejo de eventos y el comportamiento asincrónico.

¿Por qué utilizar funciones de retrollamada?

- **Operaciones asincrónicas**: las retrollamadas son cruciales para administrar operaciones asincrónicas como lectura de archivos, solicitudes de API y temporizadores.
- **Manejo de eventos**: se utilizan para responder a eventos como clics en botones, entradas de usuarios o respuestas de red.
- **Código modular**: las retrollamadas ayudan a escribir código modular y reutilizable al separar los cometidos y promover el principio de responsabilidad única.

Anatomía de una función de retrollamada

Una función de retrollamada típica tiene la siguiente estructura:

```
function funcionRetrollamada(arg1, arg2, ..., retrollamada) {
    // Realiza algunas operaciones
    // ...

    // Llama a la función de retrollamada cuando ha terminado
    retrollamada(resultado);
}
```

- **funcionRetrollamada** es la función que toma una retrollamada como argumento. Puede realizar algunas operaciones de forma asincrónica.
- Eventualmente llama a la función **retrollamada** y le pasa un resultado o un error.

Manejando errores en funciones de retrollamadas

En JavaScript, las funciones de retrollamada pueden manejar errores por convención. Es común pasar un objeto de error como primer argumento o usar el segundo argumento para representar errores. Los desarrolladores deben buscar errores y manejarlos adecuadamente dentro de la función de retrollamada.

Enfoques alternativos a las retrollamadas

1. **Promesas**: Las promesas ofrecen una forma estructurada de manejar el código asincrónico y los errores. Tienen tres estados: pendiente, cumplido y rechazado. Las promesas utilizan los métodos `.then()` y `.catch()` para manejar escenarios de éxito y error.
2. **Async/Await**: Async/await es una adición más reciente a JavaScript. Simplifica el código asincrónico al permitir a los desarrolladores escribirlo en un estilo más sincrónico. Está construido sobre Promises (el objeto en JavaScript que representa a las promesas) y es especialmente útil para manejar operaciones asincrónicas con un flujo de código más lineal.
3. **Emisores de eventos**: en Node.js, la clase `EventEmitter` le permite crear arquitecturas personalizadas basadas en eventos para manejar tareas asincrónicas.

Infierno de retrollamadas (Pirámide de retrollamadas) y ejemplo

El infierno de retrollamadas, también conocido como la "pirámide de la perdición", es un problema común en JavaScript cuando se trabaja con funciones de retrollamada profundamente anidadas. Este fenómeno ocurre cuando se encadenan múltiples operaciones asincrónicas una tras otra, lo que dificulta la lectura y el mantenimiento del código. Este documento de Markdown explica el infierno de las retrollamadas y proporciona un ejemplo sencillo.

¿Qué es un infierno de retrollamadas?

- El **infierno de retrollamadas** ocurre cuando funciones asincrónicas se anidan entre sí, lo que genera estructuras de código profundamente sangradas.
- Hace que el código sea más difícil de entender, depurar y mantener debido a niveles excesivos de sangría.
- El infierno de retrollamadas a menudo resulta del manejo secuencial de múltiples operaciones asincrónicas, como realizar solicitudes de API o leer/escribir archivos.

Ejemplo de un infierno de retrollamadas

```
operacionAsincrona1(function (resultado1) {
    // Retrollamada 1
    operacionAsincrona2(resultado1, function (resultado2) {
        // Retrollamada 2
        operacionAsincrona3(resultado2, function (resultado3) {
            // Retrollamada 3
            operacionAsincrona4(resultado3, function (resultado4) {
                // Retrollamada 4
                operacionAsincrona5(resultado4, function (resultado5) {
                    // Retrollamada 5
                    // ... etcétera
                });
            });
        });
    });
});
```

Problemas con el infierno de retrollamadas

- **Legibilidad**: el infierno de retrollamadas conduce a un código con sangría profunda, lo que dificulta su lectura y comprensión. Esto puede dificultar la revisión del código y la colaboración.
- **Mantenibilidad**: a medida que se agregan más operaciones asincrónicas, el infierno de retrollamadas hace que el código base sea difícil de mantener. Modificar la funcionalidad existente o agregar nuevas funciones se vuelve propenso a errores.

- **Manejo de errores**: la gestión de errores se vuelve compleja en las retrollamadas anidadas. Manejar excepciones y propagar errores a niveles superiores puede resultar un desafío.

Mitigar el infierno de las retrollamadas

1. Funciones nombradas

- Divida las funciones de retrollamada en funciones separadas con nombre. Esto mejora la legibilidad del código al dar nombres significativos a funciones individuales.

2. Promesas

- Las promesas proporcionan una forma más estructurada de manejar el código asincrónico. Le permiten encadenar operaciones asincrónicas, haciendo que el código sea más lineal y más fácil de leer.

3. Async/Await

- Async/await es una adición más reciente a JavaScript. Simplifica el código asincrónico al permitirle escribirlo en un estilo más sincrónico. Está construido sobre Promises y es especialmente útil para manejar operaciones asincrónicas con un flujo de código más lineal.

4. Modularización

- Organice el código en módulos más pequeños y reutilizables. Esto reduce la complejidad de las funciones individuales y facilita la gestión de operaciones asincrónicas.

Conclusión

El manejo eficaz de errores es crucial en la programación asincrónica. Las retrollamadas pueden manejar errores por convención, pero enfoques alternativos como Promises, async/await y emisores de eventos proporcionan formas más estructuradas y legibles de administrar código asincrónico. La elección de qué enfoque utilizar depende de los requisitos específicos y las preferencias de estilo de codificación.

El infierno de retrollamadas es un problema común en JavaScript cuando se trabaja con funciones de retrollamada profundamente anidadas para manejar operaciones asincrónicas. Puede generar código que sea difícil de leer, mantener y depurar. Las estrategias de mitigación, como el uso de funciones con nombre, promesas, async/await o modularización, pueden mejorar significativamente la estructura y la legibilidad del código cuando se trata de tareas asincrónicas, lo que hace que su código sea más fácil de mantener y resistente a errores.

IPA de Web y AJAX

En este capítulo, analizaremos IPA y AJAX, las dos tecnologías importantes que permiten que las aplicaciones interactúen con servidores y envíen o recuperen datos sin la necesidad de recargar la página completa.

IPA (Interfaz de Programación de Aplicaciones)

Un **IPA** (Interfaz de Programación de Aplicaciones) es un conjunto de reglas y protocolos que permite a diferentes aplicaciones de software comunicarse entre sí. Define los métodos y los formatos de datos que las aplicaciones pueden usar para solicitar e intercambiar información.

Puntos clave

- Las IPA permiten a los desarrolladores acceder a la funcionalidad de otros componentes, servicios o plataformas de software sin necesidad de comprender su funcionamiento interno.
- Las IPA proporcionan una forma para que las aplicaciones envíen solicitudes y reciban respuestas, lo que les permite interactuar y compartir datos sin problemas.
- Los tipos comunes de IPA incluyen **IPA web** que permiten que las aplicaciones web se comuniquen a través de Internet, **IPA de biblioteca** que proporcionan funciones de código reutilizables y **IPA del sistema operativo** que permiten la interacción con el sistema operativo subyacente.
- Las IPA son esenciales para crear integraciones, crear software sobre plataformas existentes y permitir la interoperabilidad entre diferentes sistemas.
- La documentación de IPA, a menudo proporcionada por desarrolladores o proveedores de servicios, explica cómo utilizar una IPA, incluidos los puntos finales disponibles, los métodos de solicitud y los formatos de respuesta.
- Ejemplos de IPA populares incluyen IPA de redes sociales (por ejemplo, IPA Graph de Facebook), IPA de pasarela de pago (por ejemplo, IPA de PayPal) y IPA de servicios en la nube (por ejemplo, IPA de AWS).

Beneficios de las IPA

- **Interoperabilidad:** Las IPA permiten que diferentes sistemas de software trabajen juntos, promoviendo la compatibilidad y el intercambio de datos.
- **Eficiencia:** Los desarrolladores pueden aprovechar las IPA existentes para ahorrar tiempo y esfuerzo, centrándose en crear funciones específicas.
- **Escalabilidad:** Las IPA permiten la expansión de servicios y funciones mediante la integración con herramientas y servicios de terceros.
- **Innovación:** Las IPA fomentan la innovación al abrir oportunidades para que los desarrolladores creen nuevas aplicaciones y servicios.
- **Seguridad:** Las IPA suelen incluir mecanismos de autenticación y autorización para garantizar el acceso seguro a datos y servicios.

AJAX (JavaScript asíncrono y XML)

AJAX, abreviatura de **JavaScript y XML asincrónico**, es una tecnología fundamental en el desarrollo web. Permite que las aplicaciones web realicen solicitudes asincrónicas a un servidor, recuperen datos y actualicen partes de una página web sin necesidad de recargar la página completa. Si bien el nombre sugiere XML, AJAX puede funcionar con varios formatos de datos, siendo JSON el más común.

¿Qué es AJAX?

En esencia, AJAX es una técnica que permite que las páginas web se comuniquen con un servidor en segundo plano, sin interrumpir la interacción del usuario con la página. Este comportamiento asincrónico se logra utilizando JavaScript y permite el desarrollo de aplicaciones web más interactivas y responsivas.

¿Cómo funciona AJAX?

1. **JavaScript**: AJAX depende en gran medida de JavaScript para iniciar solicitudes y manejar respuestas de forma asincrónica.
2. **Objeto XMLHttpRequest (XHR): si bien históricamente se usaba el objeto `XMLHttpRequest`, el desarrollo web moderno a menudo emplea la IPA `fetch` para solicitudes AJAX, lo que proporciona un enfoque más intuitivo y flexible.
3. **Comunicación del servidor**: cuando un usuario activa un evento, como hacer clic en un botón, JavaScript envía una solicitud HTTP al servidor. Estas solicitudes pueden ser GET (para recuperar datos) o POST (para enviar datos al servidor).
4. **Procesamiento asincrónico**: las solicitudes AJAX son asincrónicas, lo que significa que el navegador puede continuar ejecutando otro código mientras espera la respuesta. Esto evita que la interfaz de usuario se congele.
5. **Manejo de respuestas**: una vez que el servidor procesa la solicitud, envía una respuesta al cliente. Luego, JavaScript maneja esta respuesta, generalmente actualizando el modelo de objetos de documento (DOM) con los nuevos datos.
6. **Representación**: el contenido actualizado se representa en la página web sin necesidad de recargar la página completa, lo que resulta en una experiencia de usuario más fluida.

Beneficios de AJAX

- **Experiencia de usuario mejorada**: AJAX permite que las aplicaciones web carguen y muestren datos sin la necesidad de actualizar la página completa, lo que hace que la experiencia del usuario sea más fluida e interactiva.
- **Eficiencia**: las solicitudes AJAX son livianas y solo transfieren los datos necesarios, lo que reduce el uso del ancho de banda y mejora el rendimiento de la aplicación.
- **Actualizaciones en tiempo real**: AJAX es crucial para crear funciones en tiempo real como aplicaciones de chat, notificaciones en vivo y actualizaciones de contenido dinámico.
- **Carga dinámica**: el contenido se puede cargar según demanda, lo que permite tiempos de carga de la página inicial más rápidos y aplicaciones con mayor capacidad de respuesta.

Casos de uso comunes

Algunos escenarios comunes donde se usa AJAX incluyen:

- **Envíos de formularios**: envío de formularios sin recargar toda la página para su validación y envío de datos.
- **Desplazamiento infinito**: carga contenido adicional a medida que el usuario se desplaza hacia abajo en una página, brindando una experiencia de navegación continua.
- **Sugerencias automáticas**: proporciona sugerencias de búsqueda en tiempo real a medida que los usuarios escriben consultas de búsqueda.

- **Actualización de contenido**: actualización dinámica de contenido como noticias, información meteorológica o resultados deportivos sin necesidad de actualizar manualmente la página.

Obteniendo datos meteorológicos con la IPA OpenWeatherMap usando AJAX

En este ejemplo, demostraremos cómo usar AJAX (JavaScript asíncrono y XML) para obtener información meteorológica de la IPA OpenWeatherMap y mostrarla en una página web.

Introducción

La IPA OpenWeatherMap es una poderosa herramienta para recuperar información meteorológica de lugares de todo el mundo. Este ejemplo demuestra cómo utilizar la IPA para obtener datos meteorológicos actuales de una ciudad específica y mostrarlos en su aplicación o documentación.

Clave IPA

Antes de comenzar, debe registrarse para obtener una clave IPA de OpenWeatherMap para acceder a su IPA de datos meteorológicos. Reemplace `'SU_CLAVE_IPA'` en el código siguiente con su clave IPA real.

```
const apiKey = 'SU_CLAVE_IPA';
```

HTML de aplicación meteorológica sencilla

En este ejemplo, proporcionaremos la estructura HTML para una aplicación meteorológica sencilla que recupera y muestra datos meteorológicos de la IPA OpenWeatherMap.

Estructura HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Aplicación del tiempo</title>
</head>
<body>
  <h1>Reporte del clima</h1>
  <button id="botonBusqueda">Obtener datos</button>
  <div id="info-tiempo">
    <!-- Los datos se mostrarán aquí. -->
  </div>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

Cree un archivo JavaScript llamado `script.js` para manejar la solicitud AJAX y actualizar los datos meteorológicos en la página:

```
// Punto final y ubicación de IPA
const apiUrl = 'https://api.openweathermap.org/data/2.5/weather';
const ubicacion = 'New York'; // Reemplace con su ubicación deseada

// Obtenemos los datos del tiempo
const xhr = new XMLHttpRequest();
xhr.open('GET', `${apiUrl}?q=${ubicacion}&appid=${apiKey}`, true);

xhr.onload = function () {
  if (xhr.status === 200) {
    const infoTiempo = document.getElementById('info-tiempo');
    const datos = JSON.parse(xhr.responseText);
    const temperatura = (datos.main.temp - 273.15).toFixed(2); // Convierte de Kelvin a Celsius

    const html = `
      <p><strong>Ubicacion:</strong> ${datos.name}, ${datos.sys.country}</p>
      <p><strong>Temperatura:</strong> ${temperatura} °C</p>
      <p><strong>Tiempo:</strong> ${datos.weather[0].description}</p>
    `;

    infoTiempo.innerHTML = html;
  } else {
    const infoTiempo = document.getElementById('info-tiempo');
    infoTiempo.innerHTML = '<p>No se pudieron recuperar los datos meteorológicos.</p>';
  }
};

xhr.send();
```

Resultado

Cuando abre el archivo HTML en un navegador web, mostrará la información meteorológica para la ubicación especificada (Nueva York en este caso). Asegúrese de reemplazar `'SU_CLAVE_IPA'` con su clave IPA real de OpenWeatherMap.

Este ejemplo demuestra cómo obtener datos meteorológicos de la IPA OpenWeatherMap usando AJAX y mostrarlos en una página web simple.

Recuerde alojar sus archivos HTML y JavaScript en un servidor web si planea acceder a la IPA desde un sitio web activo.

¡Eso es todo! Ha obtenido y mostrado con éxito datos meteorológicos utilizando la IPA OpenWeatherMap y AJAX.

Conclusión

Las IPA desempeñan un papel crucial en el desarrollo de software moderno al permitir que las aplicaciones colaboren y compartan datos de forma eficaz. Comprender cómo utilizar las IPA e integrarlas en sus proyectos es fundamental para crear software interconectado y rico en funciones.

AJAX es una tecnología fundamental en el desarrollo web moderno que permite a los desarrolladores crear aplicaciones web dinámicas y responsivas. Si bien el nombre sugiere XML, AJAX es compatible con varios formatos de datos, lo que lo convierte en una herramienta versátil para mejorar la experiencia del usuario y crear aplicaciones web interactivas.

Naturaleza de un solo hilo de JavaScript

JavaScript es un lenguaje de programación de un solo subproceso que ejecuta código de forma secuencial en un subproceso principal. Se basa en patrones asincrónicos sin bloqueo para manejar tareas de manera eficiente sin bloquear el hilo principal, lo que garantiza la capacidad de respuesta en las aplicaciones web. Si bien simplifica la concurrencia, requiere un uso eficaz de retrollamadas y programación basada en eventos.

Comprensión de JavaScript de un solo subproceso

A continuación se presentan algunos puntos clave que debe comprender sobre la ejecución de un solo subproceso de JavaScript:

1. **Un subproceso, una tarea:** JavaScript opera dentro de un único subproceso de ejecución, lo que significa que solo puede realizar una tarea a la vez. Este hilo a menudo se denomina "hilo principal" o "bucle de eventos".
2. **Bloqueo versus no bloqueo:** El código JavaScript es inherentemente no bloqueante. Esto significa que cuando se encuentra una operación que requiere mucho tiempo (como una solicitud de red o lectura de un archivo), JavaScript no espera a que se complete. En cambio, delega la tarea a otra parte del entorno (por ejemplo, el navegador o el tiempo de ejecución de Node.js) y continúa ejecutando otro código.
3. **Programación asincrónica:** Para manejar operaciones que pueden consumir mucho tiempo sin bloquear el hilo principal, JavaScript depende en gran medida de patrones de programación asincrónica. Funciones como retrollamadas, promesas y async/await permiten a los desarrolladores trabajar con operaciones asincrónicas de manera efectiva.
4. **Basado en eventos:** JavaScript a menudo se describe como "basado en eventos". Esto significa que escucha y responde a eventos, como interacciones del usuario (clics, pulsaciones de teclas), temporizadores o respuestas de la red. Cuando ocurre un evento, se ejecuta la función de retrollamada correspondiente.
5. **Modelo de concurrencia:** Si bien JavaScript se ejecuta en un solo subproceso, el modelo de concurrencia permite la ejecución simultánea de código. Esto se logra mediante mecanismos como el bucle de eventos, que gestiona la ejecución de tareas asincrónicas de una manera que garantiza la capacidad de respuesta y el comportamiento sin bloqueo.
6. **Interacción entre el navegador y el entorno:** En el desarrollo web, JavaScript interactúa con el modelo de objetos de documento (DOM) del navegador y otras API del navegador. Para mantener una interfaz de usuario receptiva, el código JavaScript debe ejecutarse de manera rápida y eficiente y delegar operaciones que consumen mucho tiempo a subprocesos separados cuando sea necesario.

Ejemplo asincrónico de un solo subproceso

```
// Simular una operación asincrónica con una retrollamada
function simularOperacionAsincrona(retrollamada) {
  setTimeout(function () {
    console.log("Operación asíncrona completada.");
    retrollamada();
  }, 2000); // Simula un retardo de 2 segundos
}

console.log("Inicio del programa");

// Inicia una operación asíncrona
simularOperacionAsincrona(function () {
  console.log("Retrollamada ejecutada: manejando el resultado.");
});

console.log("Fin del programa");
```

En este ejemplo, demostramos la naturaleza de subproceso único de JavaScript y cómo maneja operaciones asincrónicas mediante retrollamadas.

Explicación del código

- Definimos una función `simularOperacionAsincrona` que simula una operación asíncrona usando `setTimeout`. Esta función toma una retrollamada como argumento, la cual se ejecutará cuando se complete la operación asíncrona.
- Iniciamos el programa escribiendo en la consola "Inicio del programa".
- Iniciamos la operación asíncrona usando `simularOperacionAsincrona`, pasándole una función de retrollamada. Esta función se ejecutará después de un retardo de dos segundos.
- Inmediatamente después de iniciar la operación asincrónica, escribimos en la consola "Fin del programa".

Flujo de ejecución

- Cuando ejecute este código, notará que aunque la operación asincrónica tarda 2 segundos en completarse, el programa no se bloquea. El mensaje "Fin del programa" se muestra inmediatamente después de iniciar la operación asincrónica, lo que demuestra el comportamiento sin bloqueo de JavaScript.
- Después del retraso de 2 segundos, la "operación asíncrona se completó". Se registra el mensaje, seguido de "Retrollamada ejecutada: manejando el resultado", lo que indica que la función retrollamada se ejecutó cuando finalizó la operación asincrónica.

Conclusiones clave

- JavaScript opera en un solo hilo y las operaciones asincrónicas se manejan mediante retrollamadas.
- La naturaleza de un solo subproceso permite que JavaScript siga respondiendo incluso durante tareas que requieren mucho tiempo.
- Las retrollamadas son un mecanismo fundamental para trabajar con código asincrónico en JavaScript.

Beneficios y desafíos

Beneficios

- Simplicidad: la ejecución de un solo subproceso simplifica el modelo de programación y reduce el riesgo de errores complejos relacionados con la concurrencia.

- Previsibilidad: la naturaleza de un solo subproceso hace que sea más fácil razonar sobre el orden de ejecución y el estado de su programa.

Desafíos

- Operaciones de bloqueo: las operaciones de larga duración pueden potencialmente bloquear el hilo principal, lo que genera una mala experiencia del usuario, especialmente en aplicaciones web.
- Infierno de retrollamadas: el uso excesivo de retrollamadas (a menudo denominado "el infierno de retrollamadas") puede hacer que el código sea más difícil de leer y mantener.
- Cuello de botella de concurrencia: las tareas vinculadas a la CPU no pueden utilizar completamente los procesadores multinúcleo porque JavaScript se ejecuta en un solo subproceso.

En resumen, la naturaleza de un solo subproceso de JavaScript es una característica definitoria del lenguaje. Si bien simplifica ciertos aspectos de la programación, también presenta desafíos en términos de manejar tareas asincrónicas y garantizar aplicaciones receptivas. El uso eficaz de patrones asincrónicos y la comprensión del modelo basado en eventos son esenciales para los desarrolladores de JavaScript.

ECMAScript (ES)

ECMAScript, comúnmente abreviado como ES, es una especificación de lenguaje de scripting estandarizado. Sirve como base para varios lenguajes de programación, siendo JavaScript la implementación más conocida y utilizada. Este documento de Markdown proporciona una descripción general de ECMAScript, su historia, características y su papel en el desarrollo web.

¿Qué es ECMAScript?

- **Lenguaje estandarizado**: ECMAScript es una especificación de lenguaje de secuencias de comandos estandarizada mantenida por ECMA International. Define la sintaxis y la semántica del lenguaje para garantizar la coherencia y la interoperabilidad.
- **Implementación de JavaScript**: JavaScript es la implementación más destacada de ECMAScript, pero otros lenguajes como ActionScript también utilizan esta especificación como base.

Historia de ECMAScript

- **ES1 (ECMAScript 1)**: Lanzado en 1997, ES1 sentó las bases para JavaScript tal como lo conocemos hoy.
- **ES3 (ECMAScript 3)**: Lanzado en 1999, ES3 introdujo mejoras significativas y se considera la versión que llevó JavaScript al desarrollo web convencional.
- **ES5 (ECMAScript 5)**: Lanzado en 2009, ES5 agregó nuevas funciones y mejoró las existentes, haciendo que JavaScript sea más sólido.
- **ES6 (ECMAScript 2015)**: Lanzado en 2015, ES6 fue un hito importante al introducir cambios significativos como funciones de flecha, clases, módulos y más.
- **ESNext**: Se refiere al desarrollo continuo de ECMAScript, donde continuamente se proponen y agregan nuevas características y mejoras.

Por qué ECMAScript (ES) está estandarizado para JavaScript

Esta sección del documento explica por qué ECMAScript es crucial para JavaScript, su papel en la estandarización y sus beneficios para el lenguaje.

La necesidad de estandarización

- **Coherencia del lenguaje**: JavaScript, como lenguaje de programación ampliamente utilizado para el desarrollo web, necesitaba una especificación estandarizada para garantizar la coherencia entre diversas implementaciones y entornos.
- **Interoperabilidad**: diferentes navegadores y motores web pueden tener sus propias interpretaciones de JavaScript. Un estándar ayuda a garantizar que el código JavaScript se comporte de manera consistente en todas las plataformas.

El papel de ECMAScript

- **Definición del lenguaje**: ECMAScript define las características principales de JavaScript, incluida su sintaxis, tipos de datos, funciones y objetos fundamentales.

- **Organismo de estandarización**: ECMAScript es mantenido y desarrollado por ECMA International (Asociación Europea de Fabricantes de Computadoras), una organización de estándares. Esta organización garantiza que JavaScript siga siendo un lenguaje estable y bien definido.
- **Evolución de la versión**: ECMAScript introduce nuevas funciones de lenguaje y mejoras en cada nueva versión, manteniendo JavaScript actualizado con las necesidades del desarrollo web moderno.

Beneficios de la estandarización de ECMAScript

- **Consistencia**: la estandarización garantiza que JavaScript se comporte de manera consistente en diferentes plataformas y navegadores, lo que reduce los problemas de compatibilidad.
- **Interoperabilidad**: los desarrolladores pueden escribir código JavaScript con confianza, sabiendo que funcionará según lo previsto en varios entornos.
- **Innovación**: el desarrollo continuo de ECMAScript permite la introducción de nuevas funciones y mejoras del lenguaje, lo que permite que JavaScript evolucione con el panorama web en constante cambio.
- **Desarrollo multiplataforma**: la estandarización facilita a los desarrolladores escribir código que funcione tanto en entornos del lado del cliente como del servidor.

Implementaciones de JavaScript

- **Navegadores principales**: los navegadores web populares como Chrome, Firefox, Safari y Edge implementan ECMAScript para ejecutar código JavaScript.
- **Node.js**: Node.js, un tiempo de ejecución de JavaScript del lado del servidor, también cumple con los estándares ECMAScript, lo que permite utilizar JavaScript para la programación del lado del servidor.

Características clave de ECMAScript

- **Funciones de flecha**: proporciona una sintaxis concisa para definir funciones y enlace léxico de `this`.
- **Clases**: Se introdujo la sintaxis de clases para la programación orientada a objetos.
- **Módulos**: Se agregó soporte nativo para importaciones y exportaciones de módulos.
- **Promesas**: Se introdujeron Promesas (los objetos Promise) para mejorar el manejo de operaciones asincrónicas.
- **Async/Await**: código asíncrono simplificado con la introducción de funciones asíncronas.
- **let y const**: variables de ámbito de bloque con `let` y constantes con `const`.
- **Desestructuración**: permite extraer fácilmente valores de matrices y objetos.
- **Literales de plantilla**: Se introdujeron literales de plantilla para una interpolación de cadenas más flexible.

El papel de ECMAScript en el desarrollo web

- **Secuencias de comandos del lado del cliente**: ECMAScript es la base para las secuencias de comandos del lado del cliente en el desarrollo web. Impulsa aplicaciones web interactivas.
- **Compatibilidad**: si bien los navegadores modernos admiten las últimas funciones de ECMAScript, los desarrolladores deben considerar la compatibilidad con versiones anteriores de navegadores más antiguos.
- **Transpiladores**: herramientas como Babel pueden transpilar código ECMAScript más nuevo a versiones anteriores para una mayor compatibilidad con el navegador.

- **TypeScript**: TypeScript, un superconjunto de ECMAScript, agrega escritura estática para mejorar las herramientas y la seguridad del código.

Conclusión

ECMAScript es una parte fundamental del desarrollo web y da forma a la forma en que creamos aplicaciones web dinámicas e interactivas. Mantenerse informado sobre las últimas funciones de ECMAScript es esencial para el desarrollo de JavaScript moderno. ECMAScript desempeña un papel crucial al proporcionar una base estandarizada para JavaScript, garantizando la coherencia, la interoperabilidad y la mejora continua del lenguaje. Esta estandarización permite a los desarrolladores escribir código JavaScript con confianza, sabiendo que funcionará de manera confiable en diferentes plataformas y entornos.

Creación e implementación de aplicaciones JS

Desarrollar e implementar una aplicación JavaScript implica una serie de pasos que van desde configurar el entorno de desarrollo hasta implementar la aplicación en un servidor web o plataforma de alojamiento. La siguiente es una guía detallada para ayudar a las personas a través de este proceso:

Configurar el entorno de desarrollo

Antes de comenzar el proceso de desarrollo, es esencial que el desarrollador se asegure de que Node.js y npm (Node Package Manager) estén instalados en su sistema. Estas herramientas vitales se pueden adquirir en el sitio web oficial de Node.js Node.js. Además, el desarrollador debe seleccionar un editor de código apropiado o un entorno de desarrollo integrado (IDE) para el desarrollo de JavaScript. Algunas de las opciones populares incluyen Visual Studio Code, Sublime Text y WebStorm.

La instalación de Node.js y npm proporciona acceso a las herramientas y bibliotecas esenciales necesarias para el desarrollo de JavaScript. La selección cuidadosa del editor de código o IDE adecuado puede mejorar sustancialmente la productividad y la calidad del código.

Elegir un marco o biblioteca de JavaScript

La elección de un marco o biblioteca de JavaScript depende de los requisitos específicos del proyecto en cuestión. Los desarrolladores pueden optar por trabajar con marcos bien establecidos como React, Angular, Vue.js, o adherirse al uso de JavaScript básico, dependiendo de la complejidad y las demandas del proyecto. La selección se guía fundamentalmente por la necesidad de una estructura y componentes prefabricados que puedan acelerar el proceso de desarrollo y reforzar la mantenibilidad.

Crear el proyecto

El inicio del proyecto se ve facilitado por la utilización de un administrador de paquetes como npm o yarn para establecer un nuevo proyecto. Por ejemplo, la ejecución del comando `npm init` se puede emplear para configurar un nuevo proyecto de Node.js. La adopción de un administrador de paquetes durante el inicio del proyecto garantiza el establecimiento de una estructura de proyecto estandarizada y agiliza la gestión de dependencias. Este enfoque ayuda significativamente a mantener la organización y la capacidad de gestión del proyecto.

Desarrollo de la aplicación

Durante todo el proceso de codificación de la aplicación JavaScript, se recomienda al desarrollador que organice diligentemente los módulos y componentes. Esta práctica es crucial para facilitar el mantenimiento en el futuro. El desarrollo de código organizado y modular es fundamental para garantizar que la aplicación siga siendo fácil de mantener y de depurar. Además, este enfoque fomenta la reutilización del código y fomenta la colaboración entre los desarrolladores que trabajan en el proyecto.

Prueba de la aplicación

Se anima al desarrollador a crear pruebas unitarias y pruebas de integración empleando marcos de prueba como Jest, Mocha o Jasmine. Esta práctica tiene como objetivo verificar que la aplicación funciona de acuerdo con los objetivos previstos. La creación de pruebas sirve como medida proactiva para identificar y abordar de forma preventiva cualquier error potencial, infundiendo así confianza en la confiabilidad de la aplicación.

Construir la aplicación

Para optimizar el código JavaScript, CSS y los activos para producción, se recomienda emplear una herramienta de compilación adecuada como Webpack, Parcel, o Rollup. Estas herramientas agrupan y optimizan código y activos, lo que lleva a tiempos de carga reducidos y un rendimiento mejorado. Además, contribuyen a la organización del código y facilitan la segregación de inquietudes dentro de la aplicación.

Configuración del despliegue

El desarrollador debe tomar una decisión bien informada con respecto a la ubicación de implementación. Las opciones de implementación abarcan alojamiento web tradicional, servicios en la nube como AWS o Google Cloud, o plataformas como Netlify, Vercel o Páginas de GitHub. La elección de la plataforma de implementación debe estar alineada con los requisitos específicos del proyecto y las restricciones presupuestarias. Las diferentes plataformas ofrecen distintos niveles de escalabilidad, seguridad y facilidad de uso.

Crear una compilación de producción

Generar una versión lista para producción de la aplicación implica ejecutar el proceso de compilación. Por lo general, esto implica la minimización y optimización del código, lo que resulta en un uso reducido del ancho de banda y una experiencia de usuario mejorada. Además, una compilación de producción garantiza que la aplicación funcione de manera óptima en entornos de producción.

Desplegar la aplicación

El proceso de implementación requiere un estricto cumplimiento de las instrucciones proporcionadas por la plataforma de alojamiento. Esto puede implicar el uso de FTP, SSH o herramientas de implementación específicas de la plataforma. Cumplir con las mejores prácticas durante la implementación es crucial para garantizar un acceso fluido de los usuarios a la aplicación. La implementación se puede lograr a través de varios medios, incluidas cargas manuales o canales de implementación automatizados.

Configuración de dominio y DNS (si corresponde)

Para aquellos que utilizan dominios personalizados, configurar los ajustes de DNS para dirigir el tráfico al proveedor de alojamiento o al servidor es un paso obligatorio. Esta configuración permite a los usuarios acceder a la aplicación a través de un nombre de dominio fácil de usar, mejorando así la marca y la accesibilidad.

Integración e implementación continuas (CI/CD)

El desarrollador puede optar por establecer un proceso de integración y despliegue continuos (CI/CD). Esto se puede lograr mediante la utilización de herramientas CI/CD como Jenkins, Travis CI, CircleCI o Acciones de GitHub. La automatización de los procesos de prueba e implementación en respuesta a cambios de código minimiza el potencial de error humano y garantiza que las alteraciones del código se sometan a pruebas rigurosas antes de llegar al entorno de producción. Este enfoque eleva significativamente la calidad y confiabilidad del código.

Monitoreo y Mantenimiento

Después de la implementación, se requiere vigilancia para monitorear la aplicación en busca de errores, problemas de rendimiento y vulnerabilidades de seguridad. La actualización periódica de las dependencias es fundamental para mejorar la seguridad y aprovechar nuevas funciones. Este enfoque proactivo garantiza que la aplicación conserve su confiabilidad, seguridad y rendimiento a lo largo del tiempo.

Escalado (si es necesario)

En escenarios donde la aplicación experimenta crecimiento y un aumento en el tráfico y la carga de trabajo, escalar la infraestructura puede resultar imperativo. Los proveedores de servicios en la nube ofrecen soluciones diseñadas para adaptarse a dichos requisitos de escala. Estas soluciones permiten que la aplicación administre sin problemas cargas elevadas mientras preserva el rendimiento y la disponibilidad.

Copia de seguridad y recuperación ante desastres (si es necesario)

La implementación de estrategias de respaldo y recuperación ante desastres es indispensable para salvaguardar los datos de la aplicación en caso de interrupciones imprevistas. Estas estrategias son fundamentales para garantizar la continuidad del negocio y mitigar el riesgo de pérdida de datos durante eventos inesperados.

Pruebas

Pruebas unitarias

Las pruebas unitarias son una práctica fundamental en el desarrollo web. Implica probar componentes o funciones individuales para garantizar que funcionen como se espera. Esta práctica puede detectar errores tempranamente, mejorar la calidad del código y hacer que la refactorización sea más segura. Las pruebas unitarias son esenciales por las siguientes razones:

- Verifica que las partes individuales de su código base estén funcionando correctamente.
- Proporciona una red de seguridad al refactorizar o realizar cambios.
- Ayuda a documentar el comportamiento esperado de funciones y componentes.

Marcos de prueba

Los marcos de prueba agilizan el proceso de redacción y ejecución de pruebas. Dos marcos populares son Jest y Mocha.

Jest

Jest es un popular marco de pruebas todo en uno y sin configuración. Es adecuado tanto para pruebas unitarias como de integración. Veamos cómo empezar con Jest.

Instale Jest usando npm o hilo:

```
npm install --save-dev jest
```

Cree un archivo de prueba (por ejemplo, `miFuncion.test.js`) para la función que desea probar.

Escriba un caso de prueba usando la función de prueba de Jest:

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

test('debe devolver la suma de dos números', () => {
  expect(miFuncion(2, 3)).toBe(5);
});
```

Ejecute pruebas usando el comando jest:

```
npx jest
```

Mocha

Mocha es un marco de prueba flexible. Proporciona la estructura para ejecutar pruebas, pero requiere bibliotecas adicionales para las aserciones y burlas.

Comenzando con Moca

Instale Mocha y una biblioteca de aserciones como Chai:

```
npm install --save-dev mocha chai
```

Create a `test` directory and add your test files.

Write your test cases using Mocha's describe and it functions and Chai's assertion functions.

```
const chai = require('chai');
const expect = chai.expect;
const myFunction = require('./myFunction');

describe('myFunction', () => {
  it('should return the sum of two numbers', () => {
    expect(myFunction(2, 3)).to.equal(5);
  });
});
```

Conclusion

In this chapter, we've explored the fundamentals of testing in web development and discussed the significance of Unit testing and other testing frameworks and tools that are vital for any web developer. With consistent practice and access to the right set of tools, one can write dependable code and ensure that applications perform optimally.

Capítulo 20

Código del lado del servidor

Código del lado del servidor se refiere al código que se ejecuta en un *servidor web* en lugar de en el navegador web de un usuario. Es responsable de procesar las solicitudes de los clientes (normalmente navegadores web) y generar páginas web dinámicas o proporcionar datos al cliente.

Código del lado del cliente se refiere al código que se ejecuta en el *navegador web* de un usuario en lugar de en un servidor web. Es responsable de generar la interfaz de usuario y manejar las interacciones del usuario. El código del lado del cliente normalmente está escrito en JavaScript y lo ejecuta el navegador.

¿Por qué necesitamos código del lado del servidor?

El código del lado del servidor es esencial en el desarrollo web por varias razones:

- **Seguridad**: el código del lado del servidor no es visible para el usuario, por lo que es más seguro que el código del lado del cliente.
- **Rendimiento**: el código del lado del servidor se puede utilizar para realizar tareas computacionalmente costosas, como el procesamiento de datos, sin afectar la experiencia del usuario.
- **Almacenamiento de datos**: el código del lado del servidor se puede utilizar para almacenar datos en una base de datos, a la que luego se puede acceder mediante el código del lado del cliente.
- **Autenticación de usuario**: el código del lado del servidor se puede utilizar para autenticar a los usuarios y restringir el acceso a ciertas partes del sitio web.
- **Contenido dinámico**: el código del lado del servidor se puede utilizar para generar páginas web dinámicas, que se pueden personalizar para cada usuario.

Del lado del servidor frente al lado del cliente

Las diferencias se resumen en la siguiente tabla:

Código del lado del servidor	Código del lado del cliente
Se ejecuta en un servidor web	Se ejecuta en un navegador web
Tiene acceso a los recursos del servidor (sistema de archivos, bases de datos, etc.).	Tiene acceso a los recursos del cliente (cookies, almacenamiento local, etc.).
Puede escribirse en una variedad de lenguajes (PHP, Python, Ruby, Java, C#, etc.).	Sólo se puede escribir en JavaScript.
Puede utilizar renderizado del lado del servidor (SSR) para generar HTML en el servidor.	Utiliza renderizado del lado del cliente (CSR) para generar HTML en el navegador.
Mejor para SEO ya que el contenido está fácilmente disponible para los motores de búsqueda.	Peor para el SEO, ya que el contenido no está disponible para los motores de búsqueda.
Puede aprovechar el almacenamiento en caché y las redes de entrega de contenido (CDN) para mejorar el rendimiento.	Control limitado sobre el almacenamiento en caché, depende del caché del navegador.

¿Por qué utilizar JavaScript para el código del lado del servidor?

A diferencia del código del lado del cliente, que sólo se puede escribir en JavaScript, el código del lado del servidor se puede escribir en una variedad de lenguajes, incluidos PHP, Python, Ruby, Java, C# y muchos más. Entonces, ¿por qué utilizar JavaScript para el código del lado del servidor? Hay varias razones:

- **Lenguaje unificado**: los desarrolladores pueden utilizar el mismo lenguaje y paradigmas de programación en toda la pila de aplicaciones, lo que puede conducir a la reutilización del código y una colaboración más sencilla entre los desarrolladores de front-end y back-end.
- **Gran ecosistema**: JavaScript tiene un vasto ecosistema de bibliotecas y paquetes disponibles a través de npm (Node Package Manager). Este rico ecosistema simplifica el proceso de desarrollo al proporcionar módulos prediseñados para diversas funcionalidades, desde enrutamiento de servidores hasta conectividad de bases de datos.
- **JSON**: La Notación de Objetos JavaScript (JSON) es un formato de datos popular que se utiliza para transmitir datos entre un servidor y una aplicación web. JSON se basa en JavaScript, por lo que es fácil trabajar con datos JSON en JavaScript.

A continuación, aprenderemos cómo usar JavaScript para el código del lado del servidor con Node.js y cómo usar Server Side Rendering (SSR) para generar HTML en el servidor.

Node.js

Node.js es un entorno de ejecución de JavaScript que permite a los desarrolladores ejecutar código JavaScript fuera de un navegador web. Está construido sobre el motor JavaScript V8, que es el mismo motor utilizado por Google Chrome. Node.js es de código abierto y multiplataforma, lo que significa que puede ejecutarse en Windows, macOS y Linux.

Node.js no es un lenguaje de programación

Mucha gente cree erróneamente que Node.js es un lenguaje de programación. Esto no es verdad. Node.js es un entorno de ejecución de JavaScript, lo que significa que proporciona un entorno para que se ejecute el código JavaScript. No es un lenguaje de programación en sí.

Node.js **no** es el único entorno de ejecución de JavaScript. Hay muchos otros, incluidos Deno, Nashorn y, más recientemente, Bun. Pero Node.js es, con diferencia, el entorno de ejecución de JavaScript más popular y utilizado.

Empezando con Node.js

Para comenzar con Node.js, deberá instalarlo en su computadora. Puede descargar la última versión de Node.js desde el sitio web oficial en nodejs.org. Una vez que haya descargado e instalado Node.js, puede verificar la instalación ejecutando el siguiente comando en su terminal:

```
node --version
```

Esto debería imprimir el número de versión de Node.js de esta manera:

```
v20.7.0
```

Escribiendo su primer programa Node.js

Ahora que ha instalado Node.js, escribamos nuestro primer programa Node.js. Cree un nuevo archivo llamado `hola.js` y agregue el siguiente código:

```
console.log('¡Hola mundo!');
```

Para ejecutar este programa, abra su terminal y navegue hasta el directorio donde guardó el archivo `hola.js`. Luego ejecute el siguiente comando:

```
node hola.js
```

Esto debería imprimir el siguiente resultado:

```
'¡Hola mundo!'
```

Escribir un servidor web simple usando Express y Node.js

Express es un marco web popular para Node.js. Proporciona una API simple y elegante para crear aplicaciones web. Usemos Express para crear un servidor web simple que responda a las solicitudes HTTP con un "¡Hola mundo!" mensaje.

Primero, necesitamos instalar Express. Para hacer esto, ejecute el siguiente comando en su terminal:

```
npm install express
```

Esto instalará Express y todas sus dependencias. Una vez que se complete la instalación, cree un nuevo archivo llamado `servidor.js` y agregue el siguiente código:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('¡Hola mundo!');
});

app.listen(3000, () => {
  console.log('El servidor está escuchando el puerto 3000');
});
```

Este código crea una nueva aplicación Express y define una ruta para la ruta raíz (`/`). Cuando se realiza una solicitud a esta ruta, el servidor responderá con un mensaje "¡Hola mundo!".

Para ejecutar este programa, abra su terminal y navegue hasta el directorio donde guardó el archivo `servidor.js` . Luego ejecute el siguiente comando:

```
node servidor.js
```

Esto debería imprimir el siguiente resultado:

```
El servidor está escuchando el puerto 3000
```

Ahora abra su navegador web y vaya a http://localhost:3000. Deberías ver un mensaje "¡Hola mundo!" .

Renderizado del lado del servidor (SSR)

Normalmente, cuando un usuario visita un sitio web, el navegador envía una solicitud al servidor, que responde con HTML, CSS y JavaScript. Pero con bibliotecas como *React* y *Vue*, el servidor solo envía una página HTML en blanco junto con un archivo JavaScript. Luego, el archivo JavaScript muestra la página en el navegador. Esto se llama **Representación del lado del cliente (CSR)**.

Server Side Rendering (SSR) es una técnica en la que el servidor procesa la solicitud y genera el HTML en el servidor a partir de los componentes React o Vue. Luego, el servidor envía el HTML generado al navegador, que luego puede representar la página sin tener que esperar a que se cargue JavaScript.

¿Porqué usar SSR?

Existen varias ventajas al utilizar SSR sobre CSR:

- **Mejor para SEO**: los motores de búsqueda pueden rastrear e indexar el contenido de su sitio web más fácilmente si se representa en el servidor. Esto puede conducir a una mejor clasificación en los motores de búsqueda y a más tráfico de los motores de búsqueda.
- **Carga de página inicial más rápida**: dado que el HTML se genera en el servidor, el navegador no tiene que esperar a que se cargue JavaScript antes de representar la página. Esto puede conducir a un tiempo de carga inicial de la página más rápido.
- **Mejor rendimiento en dispositivos de gama baja**: dado que el HTML se genera en el servidor, el navegador no tiene que trabajar tanto para representar la página. Esto puede conducir a un mejor rendimiento en dispositivos de gama baja, como teléfonos móviles y tabletas.

Desventajas de SSR

También existen algunas desventajas al usar SSR:

- **Proceso de desarrollo más complejo**: SSR requiere más trabajo en el lado del servidor, lo que puede hacer que el proceso de desarrollo sea más complejo.
- **Más recursos del servidor**: SSR requiere más recursos del servidor, lo que puede generar mayores costos de alojamiento.
- **Funcionalidad limitada del lado del cliente**: SSR no permite utilizar bibliotecas del lado del cliente, como jQuery o Bootstrap, ya que no están disponibles en el servidor.

¿Cómo implementar SSR?

Cada biblioteca tiene su propia forma de implementar SSR. Por ejemplo, para React, puede usar Next.js o Gatsby. Para Vue, puede utilizar Nuxt.js. Para Svelte puedes usar SvelteKit.

Conclusión

En este capítulo, aprendimos sobre la representación del lado del servidor (SSR) y cómo puede mejorar el rendimiento de su sitio web. También aprendimos sobre las ventajas de usar SSR sobre CSR y cómo implementar SSR con React, Vue y Svelte.

Capítulo 21

Ejercicios

En este capítulo realizaremos ejercicios para comprobar nuestro conocimiento en JavaScript. Los ejercicios que realizaremos se listan abajo: In this chapter we will be performing exercises to test our knowledge in JavaScript. The exercises that we will be performing are listed below:

- Console
- Multiplicación
- Variables de entrada del usuario
- Constantes
- Concatenación
- Funciones
- Sentencias condicionales
- Objetos
- Problema de FizzBuzz
- ¡Consigue los títulos!

Console

En JavaScript, usamos `console.log()` para escribir un mensaje (el contenido de una variable, un) In JavaScript, we use `console.log()` to write a message (the content of a variable, una cadena dada, etc.) en `console`. Se usa principalmente con fines de depuración, idealmente para dejar una traza del contenido de las variables durante la ejecución de un programa.

Ejemplo

```
console.log("Bienvenido a la edición para principiantes de Aprende JavaScript");
let edad = 30;
console.log(edad);
```

📝 Tareas

- [ ] Escriba un programa para imprimir `Hola mundo` en la consola. ¡Siéntase libre de probar otras cosas también!
- [ ] Escriba un programa para imprimir variables en la `console`.
    1. Declare una variable `animal` y asígnele el valor dragón.
    2. Imprima la variable `animal` en la `console`.

💡 Consejos

- Visite el capítulo Variables para entender más sobre variables.

Multiplicación

En JavaScript, podemos realizar la multiplicación de dos números usando el operador aritmético asterisco `(*)`.

Ejemplo

```
let valorResultante = 3 * 2;
```

Aquí, hemos almacenado el producto de `3 * 2` en la variable `valorResultante`.

📝 Tareas

- [ ] Escriba un programa que almacene el producto de `23` veces `41` e imprima su valor.

💡 Consejos

- Visite el capítulo Operadores básicos para entender las operaciones matemáticas.

Variables de entrada del usuario

En JavaScript, podemos tomar información de los usuarios y usarla como una variable. No necesitamos conocer su valor para trabajar con ésta.

📝 Tareas

- [ ] Escriba un programa que tome información de un usuario y agrege `10` a esta, e imprima su resultado.

💡 Consejos

- El contenido de una variable se determina por las entradas del usuario. El método `prompt()` salva el valor de entrada como una cadena.
- Necesitará asegurarse de que el valor de cadena se convierte en un entero para los cálculos.
- Visite el capítulo Operadores básicos para la conversión de tipos de `string` a `int`.

Constantes

Las constantes fueron presentadas en ES6(2015), y usan la palabra clave `const`. Las variables que se declaran con `const` no pueden ser reasignadas o redeclaradas.

Ejemplo

```
const VERSION = '1.2';
```

📝 Tarea

- [ ] Ejecute el programa mencionado abajo y corrija el error que ve en la consola. Asegúrese de que el resultado del código es `0.9 cuando se arregle en la consola.

```
const VERSION = '0.7';
VERSION = '0.9';
console.log(VERSION);
```

💡 Consejos

- Visite el capítulo Variables para más información sobre const y tambíen busque "*TypeError assignment to constant variable*" en los buscadores para encontrar una solución.

Concatenación

En cualquier lenguaje de programación, la concatenación de cadenas simplemente significa agregar una o más cadenas a otra cadena. Por ejemplo, cuando las cadenas "*Mundo*" y "*Buenas tardes*" se concatenan con la cadena "*Hola*", se forma la cadena "*HolaMundoBuenastardes*". Podemos concatenar una cadena de varias formas en JavaScript.

Ejemplo

```
const icono = '👋';

// usando cadenas de plantilla
`hola ${icono}`;

// usando el método join()
['hola', icono].join(' ');

// usando el método concat()
''.concat('hola ', icono);

//  usando el operador +
'hola ' + icono;

// RESULT
// hola 👋
```

📝 Tarea

- [ ] Escriba un programa que establezca los valores para `str1` y `str2` para que el código imprima '*Hola mundo*' a la consola.

💡 Consejos

- Visite el capítulo de cadenas concatenación para tener más información sobre la concatenación de cadenas.

Funciones

Una función es un bloque de código diseñado para realizar una tarea especificada y ejecutada cuando "algo" la invoque. Se puede encontrar más información sobre funciones en el capítulo funciones.

📝 Tarea

- [ ] Escriba un programa para crear una función llamada `esImpar` a la que se pasa un número `45345` como argumento y determina si el número es impar o no.
- [ ] Llame a esta función para obtener el resultaado. El resultado debería estar en un formato booleano y y debería devolver `true` en `console`.

💡 Consejos

- Visite el capítulo funciones para entender las funciones y cómo crearlas.

Sentencias condicionales

La lógica condicional es vital en programación para asegurar que el programa funciona independientemente de qué datos le arrojas y también permite la ramificación. Este ejercicio es sobre la implementación de diversas lógicas condicionales en problemas de la vida real.

📝 Tarea

- [ ] Escriba un programa para crear un mensaje "¿Cuántos km quedan por recorrer? _" y, según la respuesta del usuario y las siguientes condiciones, imprima los resultados en la `console`.
  - Si hay más de 100 km por recorrer, imprima `"Todavía te queda un poco de camino por recorrer"`.
  - Si son más de 50 km, pero menores o iguales a 100 km, imprima: `"Estaré ahí en 5 minutos"`.
  - Si son menores o iguales a 50 km, imprimir: `"Estoy estacionando. Te veo ahora mismo"`.

💡 Consejos

- Visite el capítulo lógica condicional para entender como usar la lógica condicional y las declaraciones condicionales.

Objetos

Los objetos son colecciones de pares `clave` , `valor` y cada par de clave-valor se conoce como propiedad. Aquí, la propiedad de la `clave` puede ser una `cadena` mientras que su valor puede ser de cualquier tipo.

📝 Tareas

Dada una familia Doe que incluye dos miembros, donde la información de cada miembro se proporciona en forma de un objeto.

```
let persona = {
    nombre: "John",                    //String
    apellido: "Doe",
    edad: 35,                          //Number
    genero: "masculino",
    numerosAfortunados: [ 7, 11, 13, 17], //Array
    pareja: persona2       //Object,
};

let persona2 = {
    nombre: "Jane",
    apellido: "Doe",
    edad: 38,
    genero: "femenino",
    numerosAfortunados: [ 2, 4, 6, 8],
    pareja: persona
};

let familia = {
    apellido: "Doe",
    miembros: [persona, persona2]       //Matriz de objetos
};
```

- [ ] Encuentre una forma de imprimir el nombre del primer miembro de la familia Doe en una `consola` .
- [ ] Cambien el cuarto `numerosAfortunados` del segundo miembros de la familia Doe a `33` .
- [ ] Agregue un nuevo miembro a la familia creando una nueva persona (`Jimmy,` `Doe` , `13` , `masculino` , `[1, 2, 3, 4]` , `null`) y actualize la lista de miembros.
- [ ] Imprima la `SUMA` de los números de la suerte de la familia Doe en la `consola` .

💡 Consejos

- Visite el capítulo objetos para entender cómo trabajan los objetos.
- Puede obtener los `numerosAfortunados` de cada objeto persona dentro del objeto familia.
- Una vez obtenga cada matriz simplemente itere sobre ella agregando cada elemento y después sume cada uno de los tres miembros de la familia.

Problema de FizzBuzz

El problema *FizzBuzz* es una de las preguntas más frecuentes, aquí se tiene que imprimir *Fizz* y *Buzz* bajo determinadas condiciones.

📝 Tarea

- [ ] Escriba un programa para imprimir todos los números entre 1 y 100 de tal manera que se cumplan las siguientes condiciones.
  - Para múltiplos de 3, en vez de imprimri el número, imprima `Fizz`.
  - Para múltipls de 5, imprima `Buzz`.
  - Para números que son múltiplos tanto de 3 como de 5, imprima `FizzBuzz`.

```
/
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
....
....
98
Fizz
Buzz
/
```

💡 Consejos

- Visite el capítulo loops para entender cómo trabajan los bucles.

¡Consigue los títulos!

El problema *¡Obtener los títulos!* es un problema interesante en el que tenemos que obtener el título de una lista de libros. Este es un buen ejercicio para la implementación de matrices y objetos.

📝 Tareas

Dada una matriz de objetos que representan libros con un autor.

```
const libros = [
  {
    titulo: "Eloquent JavaScript, Third Edition",
    autor: "Marijn Haverbeke"
  },
  {
    titulo: "Practical Modern JavaScript",
    autor: "Nicolás Bevacqua"
  }
]
```

- [ ] Escriba un programa que cree una función `obtenerLosTitulos` que tome la matriz y devuelva la matriz del título e imprima su valor en la `console` .

💡 Consejos

- Viste los capítulos de matrices y objetos para entender como trabajan los objetos y las matrices.

Capítulo 22

Preguntas de entrevista

Este capítulo analiza varias preguntas para preparar mejor al candidato en su comprensión de JavaScript. Se divide en tres partes: nivel básico, intermedio y avanzado.

Preguntas de entrevista sobre JavaScript de nivel básico

1. Historia y definición de variables

1.1. ¿Qué es JavaScript?

Respuesta: JavaScript es un lenguaje de programación interpretado de alto nivel que se utiliza comúnmente en el desarrollo web para agregar interactividad y comportamiento dinámico a los sitios web.

1.2. ¿Quién creo/desarrolló JavaScript?

Respuesta: JavaScript fue creado por *Brendan Eich* mientras trabajaba en **Netscape Communications Corporation**. Desarrolló el lenguaje en sólo diez días en mayo de 1995. JavaScript se llamaba originalmente "*Mocha*", pero luego pasó a llamarse "*LiveScript*" y finalmente "*JavaScript*" como parte de una colaboración de marketing con **Sun Microsystems** (ahora **Oracle Corporation**), que tenía un lenguaje de programación llamado **Java** que estaba ganando popularidad en ese momento. A pesar de la similitud de nombres, *JavaScript* y *Java* son lenguajes de programación completamente diferentes con propósitos y características distintos.

1.3. ¿Cómo declara una variable en JavaScript?

Respuesta: Puede declarar una variable usando `var`, `let`, o `const`:

- `var` (ámbito de función)
- `let` (alcance de bloque)
- `const` (con alcance de bloque, para constantes)

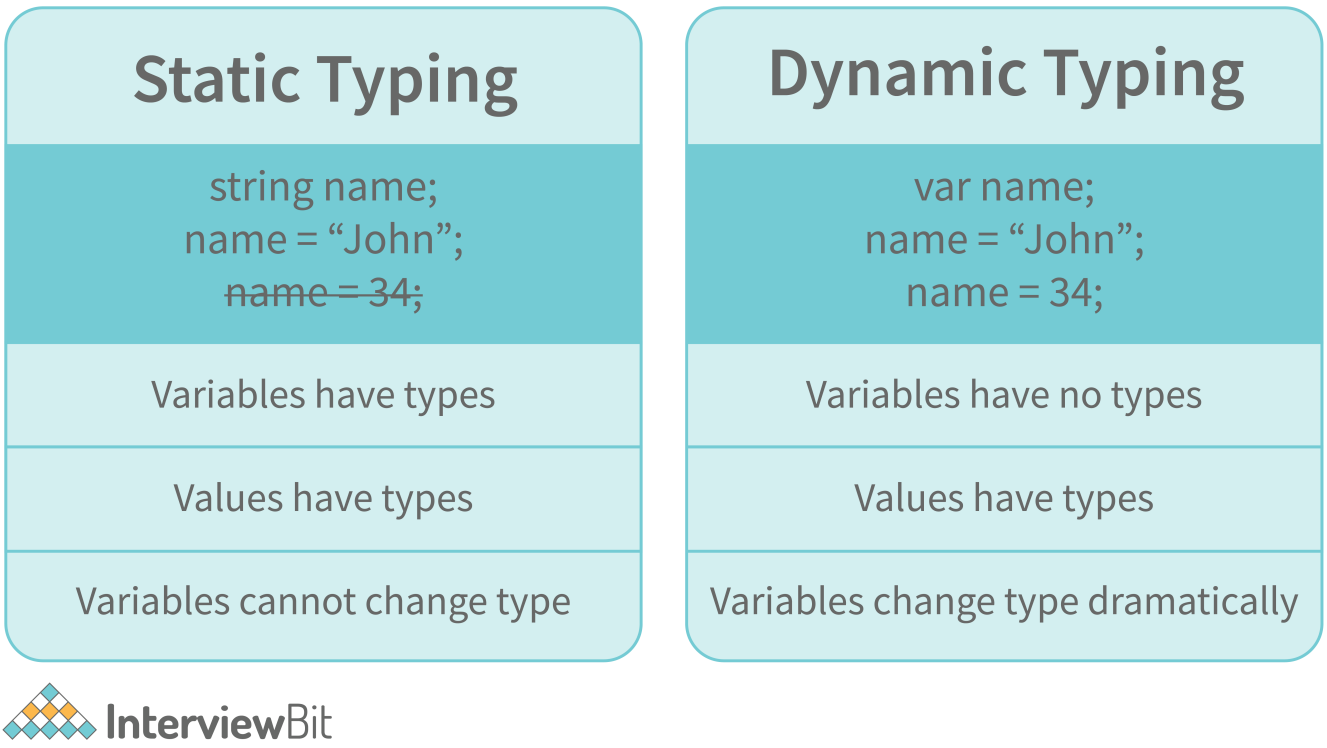
1.4. ¿Cuál es la diferencia entre `let`, `var`, y `const`?

Respuesta:

- `var` tiene ámbito de función, mientra que `let` y `const` tienen alcance de bloque.
- `let` permite la reasignación de la variable, mientras que `const` se usa para constantes.
- Las variables declaradas con `var` se elevan (hoisted, en inglés), mientras que `let` y `const` no se elevan.

1.5. ¿Es JavaScript un lenguaje de tipado dinámico o de tipado estático?

Respuesta: JavaScript es un lenguaje de tipado dinámico. En un lenguaje de tipado dinámico, el tipo de una variable se verifica durante el tiempo de ejecución, a diferencia de un lenguaje de tipado estático, donde el tipo de una variable se verifica durante el tiempo de compilación.



Dado que JavaScript es un lenguaje de *tipo debilmente acoplado (dinámico)*, las variables en JS no están asociadas con ningún tipo. Una variable puede contener el valor de cualquier tipo de datos.

Por ejemplo, una variable a la que se le asigna un tipo de número se puede convertir a un tipo de cadena:

```
var a = 23;
var a = "¡Hola mundo!";
```

1.6. ¿Cuáles son los tipos de errores en JavaScript?

Respuesta: Hay dos tipos de errores en JavaScript.

1. **Error de sintaxis**: Los errores de sintaxis son errores o problemas ortográficos en el código que hacen que el programa no se ejecute en absoluto o deje de ejecutarse a la mitad. Por lo general, también se proporcionan mensajes de error.
2. **Error lógico**: Los errores de razonamiento ocurren cuando la sintaxis es correcta pero la lógica o el programa son incorrectos. La aplicación se ejecuta sin problemas en este caso. Sin embargo, los resultados son inexactos. A veces, estos son más difíciles de corregir que los problemas de sintaxis, ya que estas aplicaciones no muestran señales de error para fallas lógicas.

1.7. Mencione algunas ventajas de JavaScript

Respuesta: Hay muchas ventajas de JavaScript. Algunos de ellos son:

- Javascript se ejecuta tanto en el lado del cliente como en el del servidor. Existe una variedad de marcos frontend que puede estudiar y utilizar. Sin embargo, si desea utilizar JavaScript en el backend, deberá aprender NodeJS. Actualmente es el único marco de JavaScript que se puede utilizar en el backend.
- Javascript es un lenguaje sencillo de aprender.
- Las páginas web ahora tienen más funciones gracias a Javascript.
- Para el usuario final, Javascript es bastante rápido.

2. Funciones

2.1. ¿Cómo crea una función en JavaScript?

Respuesta:

Puede crear una función usando la palabra clave `function` o funciones de flecha (`=>`):

Ejemplo:

```
function miFuncion() {
  // Cuerpo de la función
}

const miFuncionDeFlecha = () => {
  // Cuerpo de la función
};
```

2.2. ¿Qué son las retrollamadas?

Respuesta: Una retrollamada es una función que se ejecutará después de que se ejecute otra función. En JavaScript, las funciones se tratan como ciudadanos de primera clase, pueden usarse como argumento de otra función, pueden ser devueltas por otra función y pueden usarse como propiedad de un objeto.

Las funciones que se utilizan como argumento para otra función se denominan funciones de retrollamada. **Ejemplo**:

```
function dividirPorLaMitad(sum) {
  console.log(Math.floor(sum / 2));
}

function multiplicarPor2(sum) {
  console.log(sum * 2);
}

function operacionEnSuma(num1, num2, operacion) {
  var sum = num1 + num2;
  operacion(sum);
}

operacionEnSuma(3, 3, dividirPorLaMitad); // Muestra 3

operacionEnSuma(5, 5, multiplicarPor2); // Muestra 20
```

- En el código anterior, estamos realizando operaciones matemáticas con la suma de dos números. La función `operacionEnSuma` toma 3 argumentos, el primer número, el segundo número y la operación que se realizará en su suma (retrollamada).
- Ambas funciones `dividirPorLaMitad` y `multiplicarPor2` se utilizan como funciones de retrollamada en el código anterior.
- Estas funciones de retrollamada se ejecutarán solo después de que se ejecute la función `operacionEnSuma` .
- Por lo tanto, una retrollamada es una función que se ejecutará después de que se ejecute otra función.

2.3. Explique el alcance y la cadena de alcance en JavaScript

Respuesta: El alcance en JS determina la accesibilidad de variables y funciones en varias partes del código.

En términos generales, el alcance nos permitirá saber en una parte determinada del código cuáles son las variables y funciones a las que podemos o no acceder.

Hay tres tipos de ámbitos en JS:

- Alcance global
- Alcance Local o de Función
- Alcance del bloque

Alcance global: Las variables o funciones declaradas en el espacio de nombres global tienen alcance global, lo que significa que se puede acceder a todas las variables y funciones que tienen alcance global desde cualquier lugar dentro del código.

```
var variableGlobal = "Hola mundo";

function enviaMensaje() {
  return variableGlobal; // puede acceder a variableGlobal ya que está escrita en el espacio global
}
function enviaMensaje2() {
  return enviaMensaje(); // Puede acceder a la función enviaMensaje ya que está escrita en el espacio global
}
enviaMensaje2(); // Devuelve “Hola mundo
```

Alcance de función: Cualquier variable o función declarada dentro de una función tiene un `alcance local/de función`, lo que significa que se puede acceder a todas las variables y funciones declaradas dentro de una función desde dentro de la función y no desde fuera de ella.

```
function funcionImpresionante() {
  var a = 2;

  var multiplicaPor2 = function () {
    console.log(a * 2); // Puede acceder a la variable "a" ya que a y multiplicaPor2 están escritos dentro de la misma función
  };
}
console.log(a); // Genera un error de referencia ya que a está escrito en el ámbito local y no se puede acceder a él desde fue

multiplicaPor2(); // Lanza un error de referencia ya que multiplicarBy2 está escrito en el ámbito local
```

Alcance de bloque: El `alcance de bloque` está relacionado con las variables declaradas usando let y const. Las variables declaradas con var no tienen alcance de bloque. El alcance del bloque nos dice que cualquier variable declarada dentro de un bloque `{ }`, solo se puede acceder a ella dentro de ese bloque y no se puede acceder fuera de él.

```
{
  let x = 45;
}

console.log(x); // Da un error de referencia ya que no se puede acceder a x fuera del bloque

for (let i = 0; i < 2; i++) {
  // hace algo
}

console.log(i); // Da un error de referencia ya que no se puede acceder a i fuera del bloque de bucle for
```

Cadena de alcance: el motor JavaScript también usa el alcance para buscar variables. Entendamos eso usando un ejemplo:

```
var y = 24;

function favFunction() {
  var x = 667;
  var anotherFavFunction = function () {
    console.log(x); // No encuentra x dentro de anotherFavFunction(), por lo que busca la variable dentro de favFunction() y g
  };

  var yetAnotherFavFunction = function () {
    console.log(y); // Todavía no encuentra y dentro de AnotherFavFunction(), por lo que busca la variable dentro de favFuncti
  };

  anotherFavFunction();
  yetAnotherFavFunction();
}
favFunction();
```

Como puede ver en el código anterior, si el motor JavaScript no encuentra la variable en el ámbito local, intenta buscar la variable en el ámbito externo. Si la variable no existe en el ámbito externo, intenta encontrar la variable en el ámbito global.

Si la variable tampoco se encuentra en el espacio global, se genera un error de referencia.

2.4. Explique las funciones de orden superior en JavaScript

Respuesta:

Las funciones que operan sobre otras funciones, ya sea tomándolas como argumentos o devolviéndolas, se denominan *funciones de orden superior*.

Las funciones de orden superior son el resultado de que las funciones sean **ciudadanos de primera clase** en JavaScript.

Ejemplos de funciones de orden superior:

```
function ordenSuperior(fn) {
  fn();
}

ordenSuperior(function () {
  console.log("Hola mundo");
});
function ordenSuperior2() {
  return function () {
    return "Haz algo";
  };
}
var x = ordenSuperior2();
x(); // Devuelve "Haz algo"
```

2.5. ¿Qué quiere decir con funciones de autoinvocación en JavaScript?

Respuesta:

Sin ser solicitada, se invoca (inicia) automáticamente una expresión de autoinvocación. Si una expresión de función va seguida de (), se ejecutará automáticamente. Una declaración de función no puede invocarse por sí sola.

Normalmente, declaramos una función y la llamamos; sin embargo, se pueden usar funciones anónimas para ejecutar una función automáticamente cuando se describe y no se volverá a llamar. Y no existe un nombre para este tipo de funciones.

2.6. ¿Cuál es la diferencia entre los métodos exec() y test() en JavaScript?

Respuesta:

->test() y exec() son métodos de expresión RegExp utilizados en JavaScript. ->Usaremos exec() para buscar una cadena para un patrón específico, y si lo encuentra, devolverá el patrón directamente; de lo contrario, devolverá un resultado 'empty'. ->Usaremos una prueba() para encontrar una cadena para un patrón específico. Devolverá el valor booleano 'true' al encontrar el texto dado; de lo contrario, devolverá 'false'

3. Tipos de datos y operadores

3.1. ¿Cuáles son los diferentes tipos de datos presentes en JavaScript?

Respuesta:

1. **Tipos primitivos**

   - `String` - Representa una serie de caracteres y está escrito entre comillas. Una cadena se puede representar mediante comillas simples o dobles.

     Ejemplo :

     ```
     var str = "Vivek Singh Bisht"; //usando comillas dobles
     var str2 = "John Doe"; //usando comillas simples
     ```

   - `Number` - Representa un número y se puede escribir con o sin decimales.

     Ejemplo :

     ```
     var x = 3; //sin decimal
     var y = 3.6; //con decimal (Nota del traductor: Al contrario que en español, en inglés el separador de decimal es el
     ```

   - `BigInt` - Este tipo de datos se utiliza para almacenar números que están por encima de la limitación del tipo de datos Number. Puede almacenar números enteros grandes y se representa agregando "n" a un literal entero.

     Ejemplo :

     ```
     var bigInteger = 234567890123456789012345678901234567890n;
     ```

   - `Boolean` - Representa una entidad lógica y sólo puede tener dos valores: true (verdadero) o false (falso). Los booleanos se utilizan generalmente para pruebas condicionales.

     Ejemplo :

     ```
     var a = 2;
     var b = 3;
     var c = 2;
     (a == b)(
       // devuelve false (falso)
       a == c
     ); //devuelve true (verdadero)
     ```

   - `Undefined` - Cuando una variable se declara pero no se asigna, tiene el valor indefinido y su tipo también es indefinido.

     Ejemplo :

     ```
     var x; // El valor de x es undefined (no está definido)
     var y = undefined; // También podemos establecer el valor de una variable como undefined.
     ```

   - `Null` - Representa un valor inexistente o no válido.

Ejemplo :

```
var z = null;
```

- `Symbol` - Es un nuevo tipo de datos introducido en la versión ES6 de JavaScript. Se utiliza para almacenar un valor anónimo y único.

  Ejemplo:

```
var symbol1 = Symbol('symbol');
// typeof de los tipos primitivos :
typeof "John Doe" // Devuelve "string"
typeof 3.14 // Devuelve "number"
typeof true // Devuelve "boolean"
typeof 2345678901234567890123456789012345678901n // Devuelve bigint
typeof undefined // Devuelve "undefined"
typeof null // Devuelve "object" (una especie de error en JavaScript)
typeof Symbol('symbol') // Devuelve Symbol
```

  Los tipos de datos primitivos solo pueden almacenar un único valor. Para almacenar valores múltiples y complejos, se utilizan tipos de datos no primitivos.

2. **Tipos no primitivos**

- `Object` - Se utiliza para almacenar colecciones de datos.

  Ejemplo:

```
// Colección de datos en pares clave-valor
var obj1 = {
  x: 43,
  y: "¡Hola mundo!",
  z: function () {
    return this.x;
  },
};
```

- `Array`

  Ejemplo:

```
// Colección de datos como una lista ordenada.

var array1 = [5, "Hola", true, 4.1];
```

> **Nota: Es importante recordar que cualquier tipo de datos que no sea un tipo de datos primitivo es de tipo `Object` en JavaScript.**

3.2 Diferencia entre los operadores `==` y `===`

Respuesta: Ambos son operadores de comparación. La diferencia entre ambos operadores es que `==` se usa para comparar valores, mientras que `===` se usa para comparar valores y tipos.

Ejemplo:

```
var x = 2;
var y = "2";
let r = (x == y);
console.log(r); // Devuelve verdadero ya que el valor de x e y es el mismo

let s = (x === y);
console.log(s); // Devuelve falso ya que el tipo de x es "número" y el tipo de y es "cadena"
```

3.3. ¿Qué es la propiedad NaN en JavaScript?

Respuesta: La propiedad `NaN` representa el valor "**No es un número**". Indica un valor que no es un número legal.

`typeof` de `NaN` devolverá un Number.

Para verificar si un valor es `NaN` , usamos la función `isNaN()` ,

> Nota: la función `isNaN()` convierte el valor dado a un tipo de Number y luego equivale a NaN. **Ejemplo:**

```
isNaN("Hola"); // Devuelve true
isNaN(345); // Devuelve false
isNaN("1"); // Devuelve false, ya que  '1' se convierte al tipo Number que resulta en 0 (un número)
isNaN(true); // Devuelve false, ya que true convertido al tipo Number resulta en 1 (un número)
isNaN(false); // Devuelve false
isNaN(undefined); // Devuelve true
```

3.4. ¿Qué método se utiliza para recuperar un carácter de un índice determinado?

Respuesta: La función `charAt()` del objeto String de JavaScript encuentra un elemento de carácter en el índice proporcionado. El número de índice comienza en `0` y continúa hasta `n-1` . Aquí `n` es la longitud de la cadena. El valor del índice debe ser positivo, mayor o igual que la longitud de la cadena.

4. Algunos conceptos importantes

4.1. ¿Qué es la elevación (hoisting, en inglés) en JavaScript?

Respuesta: Hoisting es un mecanismo de JavaScript donde las variables y declaraciones de funciones se mueven a la parte superior de su alcance antes de la ejecución del código. Inevitablemente, esto significa que no importa dónde se declaren funciones y variables, se mueven a la parte superior de su alcance independientemente de si su alcance es global o local.

Ejemplo 1: Elevación de variable

```
hoistedVariable = 3;
console.log(hoistedVariable); // genera 3 incluso cuando la variable se declara después de inicializarse
var hoistedVariable;
```

Ejemplo 2: Elevación de función

```
hoistedFunction();  // Muestra "¡Hola mundo!" incluso cuando la función se declara después de llamarla

function hoistedFunction(){
  console.log("¡Hola mundo!");
}
```

Ejemplo 3: Elevación de una expresión de función

```
// La elevación también se realiza a nivel local.
function doSomething(){
  x = 33;
  console.log(x);
  var x;
}
doSomething(); // Muestra 33 ya que la variable local "x" se eleva dentro del alcance local
```

> Nota: las inicializaciones de variables no se elevan, solo se elevan las declaraciones de variables:

```
var x;
console.log(x); // Muestra "undefined" ya que la inicialización de "x" no se eleva
x = 23;
```

> Nota: para evitar el izado, puede ejecutar JavaScript en modo estricto usando `use strict` ; encima del código:

```
"use strict";
x = 23; // Da error ya que 'x' no está declarada
var x;
```

4.2. ¿Por qué usamos la palabra "debugger" en JavaScript?

Respuesta:

La palabra clave `debugger` se utiliza para crear puntos de interrupción en el código. Cuando el navegador encuentra la palabra clave depurador en el código, deja de ejecutar el código y abre la herramienta de depuración del navegador.

4.3. ¿Qué es el currying en JavaScript?

Respuesta:

Currying es una técnica avanzada para transformar una función de argumentos n en n funciones de uno o menos argumentos.

Ejemplo de una función curried:

```
function add (a) {
  return function(b){
    return a + b;
  }
}

add(3)(4)
```

Por ejemplo, si tenemos una función f (a, b), entonces la función después del currying se transformará en f(a)(b).

Al utilizar la técnica del currying, no cambiamos la funcionalidad de una función, solo cambiamos la forma en que se invoca.

Veamos el currying en acción:

```
function multiply(a,b){
  return a*b;
}

function currying(fn){
  return function(a){
    return function(b){
      return fn(a,b);
    }
  }
}

var curriedMultiply = currying(multiply);

multiply(4, 3); // Devuelve 12

curriedMultiply(4)(3); // También devuelve 12
```

Como se puede ver en el código anterior, hemos transformado la función multiplicar(a,b) en una función curriedMultiply, que toma un parámetro a la vez.

4.4. ¿Cuáles son algunas de las ventajas de utilizar JavaScript externo?

Respuesta:

JavaScript externo es el código JavaScript (script) escrito en un archivo separado con la extensión.js, y luego vinculamos ese archivo dentro del elemento <head> o <body> del archivo HTML donde se colocará el código.

Algunas ventajas del JavaScript externo son

->Permite a los diseñadores y desarrolladores web colaborar en archivos HTML y JavaScript.

->Podemos reutilizar el código.

->La legibilidad del código es simple en JavaScript externo.

Preguntas de entrevista de JavaScript de nivel intermedio

1. Bucles

1.1. ¿Cuál es la definición de iteración en un bucle de JavaScript?

Respuesta:

Una iteración en un bucle de JavaScript se refiere a cada ejecución individual del cuerpo del bucle, que normalmente corresponde a un ciclo del bucle.

1.2. ¿Cuáles son todas las estructuras de bucle en JavaScript?

Respuesta:

Bucle while: un bucle while es una declaración de flujo de control que permite que el código se ejecute repetidamente en función de una condición booleana determinada. El bucle while puede considerarse como una declaración if que se repite.

Bucle for: un bucle for proporciona una forma concisa de escribir la estructura del bucle. A diferencia de un bucle while, la declaración for consume la inicialización, la condición y el incremento/decremento en una línea, proporcionando así una estructura de bucle más corta y fácil de depurar.

Do While: un bucle do- while es similar a un bucle while con la única diferencia de que verifica la condición después de ejecutar las declaraciones y, por lo tanto, es un ejemplo de bucle de control de salida.

1.3. ¿Cómo funciona la declaración break en un bucle?

Respuesta:

La instrucción break finaliza el bucle actual o la instrucción switch y transfiere el control del programa a la instrucción que sigue a la instrucción terminada. También se puede usar para saltar más allá de una declaración etiquetada cuando se usa dentro de esa declaración etiquetada.

1.4. ¿Cómo funciona la declaración continue en un bucle?

Respuesta:

La declaración continue es una "versión más ligera" de la declaración break. No detiene todo el ciclo; en cambio, detiene la iteración actual y obliga al bucle a iniciar una nueva (si la condición lo permite).

2. Declaración switch

2.1. ¿Qué es una declaración switch en JavaScript?

Respuesta:

Una declaración switch en JavaScript es una declaración de flujo de control que evalúa una expresión y ejecuta un bloque de código específico según el caso coincidente.

2.2. ¿Cuáles son las ventajas de emplear una declaración switch?

Respuesta:

Una declaración switch puede reemplazar varias comprobaciones y es más descriptiva y más fácil de leer. Las declaraciones switch mejoran la legibilidad del código, proporcionan un mejor rendimiento, simplifican los condicionales complejos, mejoran la capacidad de mantenimiento y admiten una sintaxis más limpia.

2.3. ¿Es importante el orden de las declaraciones de caso en una declaración switch?

Respuesta:

El orden de las declaraciones de caso es importante en una declaración switch, especialmente cuando se emplea un comportamiento fallido. Los casos se evalúan secuencialmente, por lo que un caso coincidente encontrado anteriormente evitará que se prueben casos posteriores, lo que afectará la ejecución y el rendimiento.

3. Cookies de JavaScript

3.1. ¿Qué son las cookies de JavaScript?

Respuesta:

Las cookies son pequeños archivos que se almacenan en el ordenador de un usuario. Se utilizan para almacenar una cantidad modesta de datos específicos de un cliente y un sitio web en particular y se puede acceder a ellos mediante el servidor web o desde la computadora del cliente. Cuando se inventaron las cookies, eran básicamente pequeños documentos que contenían información sobre usted y sus preferencias. Por ejemplo, cuando selecciona el idioma en el que desea ver su sitio web, el sitio web guardará la información en un documento llamado cookie en su computadora, y la próxima vez que visite el sitio web, podrá leer un cookie guardada anteriormente.

3.2. ¿Cómo crear una cookie usando JavaScript?

Respuesta:

Para crear una cookie usando JavaScript solo necesita asignar un valor de cadena al objeto document.cookie.

```
document.cookie = "key1 = value1; key2 = value2; expires = date";
```

3.3. ¿Cómo leer una cookie usando JavaScript?

Respuesta:

El valor de document.cookie se utiliza para crear una cookie. Siempre que quieras acceder a la cookie puedes utilizar la cadena. La cadena document.cookie mantiene una lista de pares nombre = valor separados por punto y coma, donde nombre es el nombre de una cookie y el valor es su valor de cadena.

3.4. ¿Cómo eliminar una cookie usando JavaScript?

Respuesta:

Eliminar una cookie es mucho más fácil que crear o leer una cookie, solo necesita configurar expires = “tiempo pasado” y asegurarse de que algo defina la ruta correcta de la cookie, a menos que pocas no le permitan eliminar la cookie.

4. Diálogos emergentes en JavaScript

4.1. ¿Cuáles son los tipos de diálogos emergentes disponibles en JavaScript?

Respuesta:

Hay tres tipos de diálogos emergentes disponibles en JavaScript: Alert, Confirm, Prompt.

4.2. ¿Cuál es la diferencia entre un diálogo de alerta y un diálogo de confirmación?

Respuesta:

Un diálogo de alerta mostrará solo un botón, que es el botón Aceptar. Se utiliza para informar al usuario sobre el acuerdo que debe aceptar. Pero un diálogo de confirmación muestra dos botones Aceptar y Cancelar, donde el usuario puede decidir si está de acuerdo o no.

5. Funciones flecha

5.1. ¿Cuál es la definición de función de flecha?

Respuesta:

Una función de flecha es una sintaxis concisa para definir funciones anónimas en JavaScript, utilizando la notación de flecha. Ofrece una sintaxis más corta, un alcance léxico de `this` y no se puede utilizar como constructor.

5.2. ¿En qué se diferencian las funciones de flecha de las expresiones de funciones?

Respuesta:

Las funciones de flecha proporcionan una sintaxis más corta, no tienen sus propios argumentos this, super o new.target y no se pueden usar como constructores, a diferencia de las expresiones de función.

5.3. ¿Qué significa el enlace léxico `this` en las funciones de flecha?

Respuesta:

El enlace "Lexical this" en las funciones de flecha significa que no crean su propio contexto 'this'; en cambio, heredan 'this' de su entorno circundante, lo que reduce los problemas comunes relacionados con 'this'.

5.4. ¿Cuáles son las ventajas de utilizar funciones de flecha?

Respuesta:

Las ventajas de utilizar funciones de flecha en JavaScript incluyen una sintaxis más corta, retorno implícito y enlace léxico 'this'.

5.5. ¿Cuáles son los casos de uso comunes de las funciones de flecha?

Respuesta:

Las funciones de flecha se usan comúnmente para métodos de objetos, detectores de eventos, devoluciones de llamadas y otras funciones que requieren una sintaxis más corta y concisa.

Preguntas avanzadas de entrevista sobre JavaScript

1. Cierres y alcance

1.1. ¿Qué es un cierre en JavaScript? Proporcione un ejemplo en el que el uso de cierres pueda resultar beneficioso

Respuesta:

Un cierre en JavaScript es una función que tiene acceso a las variables de su alcance circundante, incluso después de que la función externa haya terminado de ejecutarse. Este mecanismo permite que las funciones mantengan el estado entre ejecuciones.

Ejemplo: Un uso común de los cierres es crear funciones de factoría o variables privadas. Por ejemplo, si desea generar valores de ID únicos para elementos:

1.2. ¿Cómo se relacionan los cierres con el alcance y la vida útil de las variables?

Respuesta:

Los cierres permiten que una función acceda a todas las variables, así como a las funciones, que están en su alcance léxico, incluso después de que se haya completado la función externa. Esto da como resultado que las variables se conserven en la memoria, lo que permite efectivamente que las variables tengan una vida útil prolongada en comparación con las variables locales estándar que normalmente se recolectarían como basura después de que se haya ejecutado su función principal.

2. Herencia prototípica

2.1. Explicar la diferencia entre herencia clásica y herencia prototípica

Respuesta:

La herencia clásica es un concepto que se encuentra con mayor frecuencia en los lenguajes de programación orientados a objetos tradicionales como Java o C++, donde una clase puede heredar propiedades y métodos de una clase principal. La herencia prototípica, por otro lado, es exclusiva de JavaScript. En JavaScript, cada objeto puede tener otro objeto como prototipo y puede heredar propiedades de su prototipo.

La principal diferencia es que la herencia clásica se basa en clases, mientras que la herencia prototípica se basa en objetos. Aunque ES6 introdujo la palabra clave "clase" en JavaScript, es azúcar sintáctica sobre la herencia prototípica existente.

2.2. ¿Cómo se pueden ampliar los objetos JavaScript integrados?

Respuesta:

Para ampliar los objetos JavaScript integrados, puede agregar métodos o propiedades a su prototipo. Sin embargo, generalmente no se recomienda modificar los prototipos nativos porque puede provocar problemas de compatibilidad y comportamientos inesperados, especialmente si hay cambios futuros en la especificación de JavaScript.

3. JavaScript asíncrono

3.1. Explique el bucle de eventos en JavaScript. ¿Cómo se relaciona con la pila de llamadas?

Respuesta:

El bucle de eventos es un concepto fundamental en JavaScript y es responsable de manejar la ejecución de múltiples fragmentos de su programa a lo largo del tiempo, cada uno de los cuales se ejecuta hasta su finalización. Funciona como un bucle continuo que comprueba si hay tareas esperando en la cola de mensajes. Si hay tareas y el hilo principal (pila de llamadas) está vacío, retira la tarea de la cola y la ejecuta.

La pila de llamadas, por otro lado, es una estructura de datos que rastrea la ejecución de funciones en un programa. Cuando se llama a una función, se agrega a la pila de llamadas y cuando termina de ejecutarse, se elimina de la pila.

En el contexto de JavaScript, el bucle de eventos verifica continuamente la pila de llamadas para determinar si está vacía. Si está vacío y hay funciones de devolución de llamada esperando en la cola de mensajes, esas devoluciones de llamada se ejecutan.

3.2. ¿Qué son las promesas y en qué se diferencian de las retrollamadas en la gestión de operaciones asincrónicas?

Respuesta:

Las promesas son objetos que representan la eventual finalización (o falla) de una operación asincrónica y su valor resultante. Una `Promise` se encuentra en uno de estos estados:

- `pending` : estado inicial, ni cumplido ni rechazado.
- `fulfilled` : significa que la operación prometida se ha completado y la promesa tiene un valor resultante.
- `rejected` : significa que la operación fracasó y la promesa nunca se cumplirá.

Las retrollamadas son funciones que se pasan a otra función como argumentos y se ejecutan una vez completada la función externa. Si bien tanto las promesas como las retrollamadas pueden manejar operaciones asincrónicas, las promesas proporcionan una forma más sólida de manejarlas.

Las diferencias clave incluyen:

- Las promesas permiten un mejor encadenamiento de operaciones asincrónicas.
- Las retrollamadas pueden llevar a un infierno de retrollamadas o a una pirámide fatal, donde el código se vuelve difícil de leer y administrar debido a las retrollamadas anidadas.
- Las promesas tienen un mecanismo de manejo de errores estandarizado usando `.then` y `.catch` .

3.3. Describa async/await. ¿Cómo simplifica el trabajo con código asíncrono?

Respuesta:

`async/await` es una característica sintáctica introducida en ES8 (o ES2017) para trabajar con código asíncrono de una manera más sincrónica. Permite escribir operaciones asíncronas de forma lineal sin devoluciones de llamadas, lo que genera un código más limpio y legible.

La palabra clave `async` se utiliza para declarar una función asincróna, lo que garantiza que la función devuelva una promesa. La palabra clave `await` se usa dentro de una función `async` para pausar la ejecución hasta que la promesa se resuelva o rechace.

El uso de `async/await` simplifica el manejo de errores, ya que puede usar bloques try/catch tradicionales en lugar de `.catch` con promesas.

4. Métodos de matriz avanzados

4.1. Describa las funciones de `map`, `reduce` y `filter`. Proporcione un ejemplo de un caso de uso práctico para cada

Respuesta:

- `map`: Transforma cada elemento de una matriz en función de una función, devolviendo una nueva matriz de la misma longitud. **Ejemplo:** Duplicar cada número en una matriz.

```
const numeros = [1, 2, 3, 4];
const doblados = numeros.map((num) => num * 2); // [2, 4, 6, 8]
```

4.2. ¿Cuáles son algunas limitaciones o peligros al utilizar funciones de flecha?

Respuesta: Las funciones de flecha presentan una forma concisa de escribir funciones en JavaScript, pero vienen con ciertas limitaciones:

1. **Sin enlace `this`**: las funciones de flecha no vinculan su propio `this`. Heredan el enlace `this` del ámbito circundante. Esto puede resultar problemático, especialmente cuando se usan como métodos en objetos o como controladores de eventos.
2. **Sin objeto `arguments`**: Las funciones de flecha no tienen su propio objeto `arguments`. Si necesita acceder al objeto arguments, deberá utilizar expresiones de funciones tradicionales.
3. **No se pueden usar como constructores**: Las funciones de flecha no se pueden usar como constructores con la palabra clave `new` porque no tienen el método interno `[[Construct]]`.
4. **Sin propiedad `prototype`**: a diferencia de las funciones normales, las funciones de flecha no tienen una propiedad `prototype`.
5. **Menos legible para lógica compleja**: para operaciones simples, la sintaxis concisa es beneficiosa. Sin embargo, para funciones que contienen lógica compleja, la sintaxis concisa puede hacer que el código sea menos legible.

5. Palabra clave "this"

5.1. Explique el comportamiento de la palabra clave `this` en diferentes contextos, como en un método, una función independiente, una función de flecha y un controlador de eventos

Respuesta: La palabra `this` puede variar según el contexto en el que se utilice. Algunos de ellos se exploran a continuación:

- En un método: Se refiere al objeto sobre el que se invoca el método.

```
const persona = {
  nombre: "Alicia",
  diHola: function () {
    console.log(`Hola, mi nombre es ${this.nombre}`);
  },
};

persona.diHola(); // Salida: Hola, mi nombre es Alicia
```

- En función independiente: aquí, el comportamiento de `this` depende de cómo se llama a la función. Si la función se llama en el ámbito global, `this` se refiere al objeto global.

```
function saludo() {
  console.log(`Hola, mi nombre es ${this.nombre}`);
}

const nombre = "Alicia";
saludo(); // Salida: Hola, mi nombre es Alicia
```

- En una función flecha: las funciones flecha capturan este valor de su alcance léxico circundante, a diferencia de las funciones normales. Esto significa que carecen de su propio contexto.

```
const persona = {
  nombre: "Roberto",
  diHola: () => {
    console.log(`Hola, mi nombre es ${this.nombre}`);
  },
};

persona.diHola(); // Salida: Hola, mi nombre es undefined
```

5.2. ¿Cómo puede asegurarse de que una función utilice un objeto específico como su valor `this`?

Respuesta: Podemos asegurarnos de que una función use un objeto específico como su valor `this` en JavaScript usando métodos como bind(), funciones flecha, call(), apply() o definiendo métodos dentro de las clases de ES6. Estas técnicas nos permiten controlar el contexto en el que opera la función y garantizar que acceda a las propiedades y métodos del objeto deseado.

6. Gestión de la memoria

6.1. ¿Qué son las pérdidas de memoria en JavaScript? Describa las posibles causas y cómo prevenirlas

Respuesta: Las pérdidas de memoria en JavaScript ocurren cuando el programa retiene involuntariamente referencias a objetos que ya no son necesarios, lo que genera un mayor uso de memoria y posibles problemas con la aplicación. Las causas comunes incluyen variables no utilizadas, cierres, detectores de eventos y referencias circulares. Para evitar pérdidas de memoria, los desarrolladores deben eliminar referencias explícitamente, administrar detectores de eventos, evitar dependencias circulares, usar referencias débiles, emplear herramientas de creación de perfiles de memoria, realizar pruebas y revisiones de código, y utilizar linters y herramientas de análisis estático para detectar problemas potenciales en las primeras etapas del proceso de desarrollo.

6.2. Describa la diferencia entre copia superficial y copia profunda. ¿Cómo puedes lograr cada una en JavaScript?

Respuesta: La copia superficial y la copia profunda son métodos para duplicar objetos o matrices en JavaScript.

La copia superficial duplica la estructura y los valores de nivel superior de un objeto o matriz, pero conserva las referencias a objetos o matrices anidados. Los cambios en estructuras anidadas afectan tanto al original como a la copia. La copia profunda crea un nuevo objeto o matriz y duplica recursivamente todos los niveles de estructuras anidadas, asegurando que los cambios en la copia no afecten al original. Para lograr una copia superficial, podemos usar métodos como el operador de extensión o slice(). Para una copia profunda, se necesitan lógica personalizada o bibliotecas como cloneDeep de lodash debido a la falta de métodos de copia profunda integrados en JavaScript.

7. ES6 y más allá

7.1. Explique el propósito y el uso de la asignación de desestructuración de JavaScript

Respuesta: La asignación de desestructuración de JavaScript es una característica que simplifica la extracción de valores de objetos y matrices, haciendo que el código sea más conciso y legible. Nos permite asignar valores a variables en función de los nombres de propiedad (desestructuración de objetos) o posición (desestructuración de matrices). La desestructuración también se puede utilizar en los parámetros de funciones y admite la sintaxis restante para capturar los elementos restantes. Es una herramienta poderosa para trabajar con estructuras de datos complejas.

7.2. Describa la importancia de los módulos JavaScript y la sintaxis `import/export` de ES6

Respuesta: Los módulos de JavaScript, junto con la sintaxis de import/export de ES6, son cruciales para el desarrollo de JavaScript moderno. Permiten a los desarrolladores organizar, reutilizar y mantener el código de forma eficaz. Los módulos encapsulan código relacionado, promueven la reutilización del código, administran dependencias y mejoran la escalabilidad del código. La sintaxis de import/export de ES6 proporciona una forma estandarizada de declarar y usar módulos, lo que facilita estructurar y compartir código de una manera limpia y fácil de mantener. Estas características se han vuelto esenciales para crear aplicaciones JavaScript modulares y fáciles de mantener, tanto en el lado del cliente como del servidor.

7.3. ¿Cómo mejoran los literales de plantilla la manipulación de cadenas en ES6? Proporcione ejemplos

Respuesta: Los literales de plantilla en ES6 mejoran la manipulación de cadenas al permitir a los desarrolladores crear cadenas con expresiones incrustadas y contenido multilínea de una manera más legible y flexible. Admiten interpolación de variables, cadenas multilínea, evaluación de expresiones, llamadas a funciones e incluso casos de uso avanzados como plantillas etiquetadas. Esta característica mejora la legibilidad y el mantenimiento del código cuando se trabaja con cadenas complejas que involucran contenido o expresiones dinámicas.

8. Programación funcional

8.1. ¿En qué se diferencia la programación funcional de la programación imperativa en JavaScript?

Respuesta:

La programación funcional y la programación imperativa son dos paradigmas de programación predominantes.

- **Programación imperativa**: este paradigma consiste en decirle a la computadora "cómo" hacer algo y se basa en declaraciones que cambian el estado de un programa. En esencia, se centra en describir los pasos para lograr una tarea particular. Esto suele implicar bucles, condicionales y declaraciones que modifican variables.

```
let total = 0;
for(let i = 0; i < array.length; i++) {
    total += array[i];
}```
```

- **Programación funcional (FP)**

La FP se trata más de indicarle a la computadora "qué" lograr, en lugar de detallar "cómo" lograrlo. Trata las tareas computacionales como evaluaciones de funciones matemáticas y evita datos mutables y alteraciones de estado. En el contexto de JavaScript y la mayoría de los lenguajes FP:

1 **Funciones puras**: Estas son funciones donde el valor de salida está determinado únicamente por sus valores de entrada, sin efectos secundarios observables. Esto significa que, para la misma entrada, la función siempre producirá la misma salida.

2 **Datos inmutables**: una vez que se crean los datos, nunca pueden cambiar. En lugar de alterar los datos existentes, las prácticas de programación funcional implican la creación de nuevas estructuras de datos.

3 **Funciones de Primera Clase y de Orden Superior**: En FP, las funciones son ciudadanos de primera clase. Esto significa que pueden asignarse a variables, pasarse a otras funciones como parámetros y devolverse como valores. Una función de orden superior es una función que recibe otra función como argumento, devuelve una función o ambas cosas.

8.2. Explique las funciones de primera clase y su importancia en la programación funcional

Respuesta:

En JavaScript y muchos otros lenguajes de programación, las funciones se consideran "ciudadanos de primera clase". Esto significa que las funciones pueden ser:

- Asignado a variables.
- Pasado como argumentos a otras funciones.
- Devueltos de otras funciones como valores.
- Almacenado en estructuras de datos como matrices y objetos.

Aquí hay un ejemplo simple que demuestra estas propiedades:

```
// Asignando una función a una variable
const greet = function(name) {
return "Hello, " + name;
}

// Pasando una función como un argumento a otra función
function runFunction(fn, value) {
return fn(value);
}
runFunction(greet, 'John'); // Devuelve: "Hello, John"

// Devolviendo una función desde otra función
function multiplier(factor) {
  return function(number) {
    return number \* factor;
  }
}

const double = multiplier(2);
double(5); // Devuelve: 10

// Almacenando una función en una matriz
const functions = [greet, double];
```

Capítulo 23

Patrones de diseño

Los patrones de diseño son soluciones orientadas a objetos que puede implementar para resolver problemas de programación comunes que pueden ocurrir durante el proceso de desarrollo. Los patrones de diseño no son lo mismo que los algoritmos, son un concepto de codificación que puede utilizar para resolver tipos específicos de problemas.

Hay un total de 23 patrones de diseño, estos no son exclusivos de ningún idioma. Estos son conceptos fundamentales que se pueden aplicar en una amplia variedad de lenguajes de programación. Hoy aprenderemos sobre cada patrón de diseño y cómo implementarlos usando javascript. Los patrones de diseño se pueden clasificar en una de tres categorías.

- Patrones creacionales
- Patrones estructurales
- Patrones de comportamiento

Patrones de diseño creacionales

Los patrones de diseño creacional se centran en los mecanismos de creación de objetos.

1. Método Factoría

Una función de factoría es simplemente una función que crea un objeto y lo devuelve. Es un patrón de diseño creacional que le permite crear objetos sin especificar la clase o constructor exacto que se utilizará. Centraliza la lógica de creación de objetos, lo que permite flexibilidad en la creación de diferentes tipos de objetos. Digamos que tienes un sitio web y quieres crear un método que te permita crear fácilmente objetos html y agregarlos al DOM. Una factoría es la solución perfecta para esto y así es como podemos implementarla.

1.1. Componentes del método Factoría

- *Creador*

  Este es el método implementado en la factoría que crea nuevos productos.

- *Producto Abstracto*

  Una interfaz para el producto que se está creando.

- *Producto concreto*

  Este es el objeto real que se está creando.

1.2. Beneficios del método Factoría

- **Abstracción de la creación de objetos**

Elimina la complejidad de crear un objeto, permitiendo que el código del cliente se centre únicamente en los objetos creados.

- **Flexibilidad y personalización**

Las factorías permiten la personalización del proceso de creación de objetos, permitiendo variaciones en los objetos creados.

- **Encapsulación de la lógica de la creación**

La lógica de creación está encapsulada dentro de la factoria, lo que facilita modificar o ampliar el proceso de creación sin afectar el código del cliente.

- **Creación de objetos complejos**

Las fábricas son útiles cuando la creación de objetos es compleja, implica múltiples pasos o requiere que se cumplan ciertas condiciones.

1.3. Ejemplo

```
    function elementoFactoria(tipo, texto, color){
        const nuevoElemento = document.createElement(tipo)
        nuevoElemento.innerText = texto
        nuevoElemento.style.color = color
        document.body.append(nuevoElemento)


    function setText(nuevoTexto) {
         nuevoElemento.innerText = nuevoTexto;
    }

    function setColor(nuevoColor) {
        nuevoElemento.innerText = nuevoColor;
    }

    return {
        nuevoElemento,
        setText,
        setColor,
        }
    }

const h1Tag = elementoFactoria('h1','Texto inicial','Blue');

h1Tag.setText('Hola mundo');

h1Tag.setColor('Red');
```

2. Método de factoría abstracta

Las factorías abstractas son otro patrón de diseño creacional. Su objetivo principal es proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Este patrón garantiza que los objetos creados sean compatibles y funcionen juntos.

2.1. Los 4 componentes de una factoría abstracta

- *Factorías abstractas*

Esto define la interfaz para crear productos abstractos, que son familias de objetos relacionados (por ejemplo, componentes de interfaz de usuario). La fábrica abstracta declara métodos de creación para cada tipo de producto de la familia.

- *Factorías concretas*

Estas son clases que implementan la interfaz de factoría abstracta, proporcionando implementaciones específicas para crear productos concretos. Cada factoría concreta crea una familia de productos relacionados (por ejemplo, la factoría de UI puede crear un botón o una casilla de verificación).

- *Productos abstractos*

Estas son las interfaces o clases base para los productos que crea la factoría abstracta. Cada tipo de producto de la familia tiene su propia definición de producto abstracta (por ejemplo, Boton, CasillaDeVerificacion).

- *Productos concretos*

Estas son las implementaciones reales de los productos abstractos. Cada factoría concreta crea su propio conjunto de productos concretos. Los productos concretos implementan las interfaces de productos abstractas definidas para su familia (por ejemplo, HTMLButton, WindowsButton).

2.2. Beneficios de una factoría abstracta

- **Consistencia** Garantiza que los objetos creados sean compatibles y sigan un tema o estilo coherente.
- **Aislamiento de responsabilidades** Aísla la creación de objetos del código del cliente, promoviendo una clara separación de cometidos.
- **Flexibilidad y escalabilidad** Permite agregar fácilmente nuevas familias de productos sin modificar el código de cliente existente.

2.3. Ejemplo

```
 // Factoría abstracta: UIFactoria
class UIFactoria {
    creaBoton() {
        throw new Error('el método creaBoton debe ser sobreescrito');
    }

    creaCasillaVerificacion() {
        throw new Error('el método creaCasillaVerificacion debe ser sobreescrito');
    }
}

// Factoría concreta: WindowsUIFactoria
class WindowsUIFactoria extends UIFactoria {
    creaBoton() {
        return new WindowsBoton();
    }

    creaCasillaVerificacion() {
        return new WindowsCasillaVerificacion();
    }
}

// Concrete Factory: MacUIFactoria
class MacUIFactoria extends UIFactoria {
    creaBoton() {
        return new MacBoton();
    }

    creaCasillaVerificacion() {
        return new MacCasillaVerificacion();
    }
}

// Producto abstracto: Boton
class Boton {
    reproducir() {
        throw new Error('el método reproducir debe ser sobreescrito');
    }
}

// Producto concreto: WindowsBoton
class WindowsBoton extends Boton {
    reproducir() {
        console.log('reproduciendo un botón de Windows');
    }
}

// Producto concreto: MacBoton
class MacBoton extends Boton {
    reproducir() {
        console.log('reproduciendo un botón de Mac');
    }
}

// Producto abstracto: CasillaVerificacion
class CasillaVerificacion {
    reproducir() {
        throw new Error('el método reproducir debe ser sobreescrito');
    }
}

// Concrete Product: WindowsCasillaVerificacion
class WindowsCasillaVerificacion extends CasillaVerificacion {
    reproducir() {
        console.log('reproduciendo una casilla de verificación de Windows');
    }
}
```

```
// Concrete Product: MacCasillaVerificacion
class MacCasillaVerificacion extends CasillaVerificacion {
    reproducir() {
        console.log('reproduciendo una casilla de verificación de Mac');
    }
}

// Usage
const windowsFactoria = new WindowsUIFactoria();
const macFactoria = new MacUIFactoria();

const botonWindows = windowsFactoria.creaBoton();
botonWindows.reproducir();  // Salida: reproduciendo un botón de Windows

const casillaVerificacionMac = macFactoria.creaCasillaVerificacion();
casillaVerificacionMac.reproducir();  // Salida: reproduciendo una casilla de verificación de Mac
```

3. Constructor

El objetivo de un constructor es separar la construcción de un objeto de su representación. Lo que hace el patrón de construcción es básicamente permitir que el cliente construya un objeto complejo simplemente pasando el tipo y el contenido del objeto únicamente. El cliente no tiene que preocuparse por los detalles constructivos.

3.1. Los 4 componentes de un constructor

- *Constructor* El constructor suele contener una serie de métodos para construir varias partes del objeto.
- *Constructor concreto* Implementa métodos desde la interfaz del constructor para construir partes del objeto.
- *Director (Opcional)* Esto no siempre es necesario, pero puede ayudar a construir el objeto final mediante un proceso de construcción específico.
- *Objeto* Representación del producto final. Contiene piezas que fueron construidas por el constructor.

3.2. Beneficios del patrón constructor

- **Separación de responsabilidades** El patrón Constructor separa la construcción de un objeto complejo de su representación, lo que permite que diferentes implementaciones de constructores varíen la representación interna.
- **Creación de objetos flexibles** Permite la creación de diferentes configuraciones de un objeto complejo mediante el uso de un proceso de construcción común. Los constructores pueden adaptarse para crear variaciones específicas del objeto.
- **Legibilidad mejorada** El uso de un constructor puede mejorar la legibilidad del código al describir claramente los pasos necesarios para construir un objeto. Es fácil entender qué aporta cada paso al objeto final.
- **Construcción parametrizada** Los constructores le permiten construir un objeto pasando parámetros a los pasos de construcción, lo que permite un control detallado sobre la creación y configuración del objeto.
- **Reutilización** Los constructores se pueden reutilizar para crear múltiples instancias del objeto complejo con diferentes configuraciones, promoviendo la reutilización del código y minimizando la duplicación de la lógica de construcción.

3.3. Ejemplo

```
//Constructor
class ComputadoraConstructor {
    construyeCPU() {}
    construyeRAM() {}
    construyeAlmacenamiento() {}
    obtenResultado() {}
}

//Constructores concretos
class ComputadoraJuegosConstructor extends ComputadoraConstructor {
    // Implementa pasos específicos para construir una computadora para juegos
}

class ComputadoraOficinaConstructor extends ComputadoraConstructor {
    // Implementa pasos específicos para construir una computadora de oficina.
}

//Clase Objeto
class Computadora {
    constructor() {
        this.piezas = [];
    }

    agregaPieza(pieza) {
        this.piezas.push(pieza);
    }
}

// Director (Opcional)
class EnsambladorComputadora {
    constructor(computadoraConstructor) {
        this.computadoraConstructor = computadoraConstructor;
    }

    ensamblaComputadora() {
        this.computadoraConstructor.construyeCPU();
        this.computadoraConstructor.construyeRAM();
        this.computadoraConstructor.construyeAlmacenamiento();
        return this.computadoraConstructor.obtenResultado();
    }
}
```

4. Único (más conocido como Singleton, en inglés)

Un único es un objeto del que solo se puede crear una instancia una vez. Los únicos son útiles cuando es necesario coordinar acciones de todo el sistema desde un único lugar central. Los únicos reducen la necesidad de variables globales, lo cual es particularmente importante en javascript porque limita la contaminación del espacio de nombres.

4.1. Componentes de Único

- *Función anónima* Un singleton se implementa utilizando una función anónima.
- *Función getInstance* Esta es una función que devuelve el objeto instanciado único.
- *Constructor (Opcional)* En javascript, no es necesario un constructor para implementar el patrón Único, pero tener un constructor es común porque le permite configurar el Único y agregar lógica de inicialización.

4.2. Beneficios del patrón Único

- **Reduce variables globales** Los únicos pueden ayudar a reducir la cantidad de variables globales requeridas en su programa, promoviendo una mejor organización y mantenibilidad del código.

- **Ahorro de memoria** Debido a que un Único garantiza que solo exista una instancia a la vez, se ahorra memoria porque evita tener múltiples instancias de la misma clase.
- **Punto de acceso global** Los únicos proporcionan un punto global de acceso a la instancia. Esto permite que otras partes del programa accedan y utilicen la misma instancia sin necesidad de pasarla.
- **Intercambio de recursos** Los únicos son especialmente útiles cuando se trata de tareas como gestionar recursos compartidos. Los únicos se pueden utilizar para administrar conexiones de bases de datos, controladores de archivos e incluso grupos de subprocesos, lo que garantiza que estos recursos se compartan de manera eficiente en toda la aplicación.

4.3. Ejemplo

```
class Unico {
  constructor() {
    const variablePrivada = 'Esto es una variable privada';

    function metodoPrivado() {
      console.log('Este es un método privado.');
    }

    return {
      metodoPublico: function() {
        console.log('Este es un método público.');
      },
      variablePublica: 'Soy pública'
    };
  }

  static getInstance() {
    if (!Unico.instance) {
      Unico.instance = new Unico();
    }
    return Unico.instance;
  }
}

const unicoInstancia1 = Unico.getInstance();
const unicoInstancia2 = Unico.getInstance();

console.log(unicoInstancia1 === unicoInstancia2); // Salida: true
```

5. Prototipo

El patrón prototipo es una forma alternativa de implementar la herencia, pero la principal diferencia es que, en lugar de heredar propiedades de una clase, los objetos heredan propiedades de un objeto prototipo. El patrón de prototipo también se conoce como patrón de propiedades y Javascript tiene soporte nativo para prototipos. En Javascript, cada objeto tiene un prototipo (referencia a otro objeto). Cuando intenta acceder a una propiedad que no existe en el objeto mismo, Javascript la buscará en el prototipo del objeto y continuará por la cadena del prototipo hasta que la encuentre o llegue al final de la cadena.

5.1. Componentes del patrón Prototipo

- *Objeto prototipo* Contiene las propiedades y métodos que heredarán todas las nuevas instancias.
- *Cliente* El cliente es responsable de crear nuevos objetos basados en el prototipo. El cliente puede crear nuevas instancias basadas en el prototipo y modificar sus propiedades en consecuencia.
- *Mecanismo de clonación/creación* El mecanismo utilizado para crear un nuevo objeto basado en el prototipo. En Javascript esto se puede lograr usando la función `Object.create()`.

5.2. Beneficios del patrón Prototipo

- **Creación de instancias eficiente**

Crear nuevas instancias del Prototipo es más eficiente que usar clases y constructores tradicionales. Los objetos se crean clonando el prototipo, lo que reduce la necesidad de configurar clases y lógica de inicialización.

- **Reutilización del código**

El patrón Prototipo le permite definir un conjunto de propiedades y métodos predeterminados en un objeto prototipo. Esto permite que varias instancias compartan el mismo comportamiento y estructura sin duplicar el código. Esto también reduce el uso de memoria ya que cada instancia no necesita almacenar duplicados de las propiedades de los prototipos.

- **Creación de objetos flexibles**

Los objetos creados con el patrón Prototipo se pueden personalizar fácilmente modificando sus propiedades o agregando nuevas propiedades específicas de la instancia.

- **Cambios dinámicos en tiempo de ejecución**

Los cambios realizados en el objeto prototipo en tiempo de ejecución se reflejan en todas las instancias basadas en el prototipo. Este comportamiento permite actualizaciones y modificaciones del prototipo, lo que afecta a todas las instancias que comparten el mismo prototipo.

5.3. Ejemplo

```
const camaraPrototipo = {
    modelo: 'predeterminado',
    marca: 'predeterminada',
    disparador: function () {
        console.log(`La ${this.marca} ${this.modelo} ha tomado una foto`);
    }
};

const camara1 = Object.create(camaraPrototipo);
camara1.modelo = 'X-Pro 3';
camara1.marca = 'Fujifilm';

const camara2 = Object.create(camaraPrototipo);
camara2.modelo = 'R5';
camara2.marca = 'Canon';
```

Patrones de diseño estructural

Se centra en cómo se componen las clases y los objetos para formar estructuras más grandes.

1. Adaptador

El Adaptador es un patrón de diseño estructural que le permite hacer que diferentes interfaces con diferentes métodos funcionen juntas sin cambiar su código. El propósito de un Adaptador es hacer que dos interfaces incompatibles funcionen juntas sin problemas.

1.1 Componentes del patrón adaptador

- *Clase/interfaz Objetivo* Esta es la interfaz o clase con la que trabajará el cliente. Contiene todos los métodos y propiedades que utilizará el código del cliente.
- *Adaptee* Adaptee es la antigua interfaz/clase que contiene propiedades y métodos que son incompatibles con la nueva interfaz/clase.
- *Adaptador* El Adaptador es lo que cierra la brecha entre el Adaptee y la interfaz/clase objetivo.

1.2. Beneficios del patrón adaptador

- **Fácil integración** Los adaptadores crean una manera fácil para que nuevos códigos o sistemas interactúen con los existentes. Al utilizar adaptadores, la integración de código nuevo se vuelve más fluida y menos propensa a errores.
- **Compatibilidad y reutilización** Los adaptadores promueven la reutilización del código y amplían la usabilidad del código existente al hacer que el código antiguo sea compatible con el código más nuevo.
- **Integración gradual del sistema** En situaciones en las que es necesario implementar un nuevo sistema de forma gradual, los adaptadores pueden actuar como intermediarios, permitiendo que nuevas funciones lleguen lentamente mientras se mantiene la compatibilidad con el sistema existente.
- **Capacidad de prueba mejorada** Los adaptadores facilitan las pruebas al permitir burlarse o eliminar al adaptee durante la prueba del código del cliente. Esto mejora la capacidad de prueba del código del cliente y ayuda a escribir pruebas unitarias más comprensibles.

1.3. Ejemplo

```
// Adaptee: conector de carga EU
class EUCConectorCarga {
  cargaConConectorEU() {
    console.log('Carga con enchufe de la UE');
  }
}

// Adaptee: conector de carga de EEUU
class USConectorCarga {
  cargaConConectorUS() {
    console.log('Carga con enchufe de EEUU');
  }
}

// Objetivo: Interfaz de carga común esperada por el cliente
class InterfazCarga {
  carga() {
    console.log('Cargando...');
  }
}

// Adaptador para el conector de carga de la UE
class EUCargaAdaptador extends InterfazCarga {
  constructor(euConectorCarga) {
    super();
    this.euConectorCarga = euConectorCarga;
  }

  carga() {
    this.euConectorCarga.cargaConConectorEU();
  }
}

// Adaptador para el conector de carga de EEUU
class USCargaAdaptador extends InterfazCarga {
  constructor(usConectorCarga) {
    super();
    this.usConectorCarga = usConectorCarga;
  }

  carga() {
    this.usConectorCarga.cargaConConectorUS();
  }
}

// Cliente
function cargarDispositivo(interfazCarga) {
  interfazCarga.carga();
}

// Uso
const euCConectorCarga = new EUCConectorCarga();
const euAdaptador = new EUCargaAdaptador(euCConectorCarga);

const usConectorCarga = new USConectorCarga();
const usAdaptador = new USCargaAdaptador(usConectorCarga);

console.log('Carga con bloque de carga de la UE:');
cargarDispositivo(euAdaptador);

console.log('Carga con bloque de carga estadounidense:');
cargarDispositivo(usAdaptador);
```

2. Puente

El Puente es un patrón de diseño estructural diseñado para dividir una clase muy grande en dos jerarquías separadas que pueden desarrollarse de forma independiente. Las dos jerarquías se denominan nivel de abstracción y nivel de implementación. Básicamente, si tiene una clase que tiene múltiples variantes de alguna funcionalidad, puede usar un patrón Bridge para dividir y organizar la clase en dos jerarquías más fáciles de entender.

2.1. Componentes del patrón Puente

- *Abstracción* Esta es la interfaz o abstracción de alto nivel. Define la funcionalidad abstracta que utilizarán los clientes.
- *Abstracción refinada* Son subclases o extensiones de la capa de abstracción. Estos proporcionan características o mejoras adicionales. Amplía la funcionalidad definida por la abstracción.
- *Implementador* Esta es la interfaz que define los métodos de implementación. Por lo general, no refleja la interfaz de abstracción, pero es una interfaz de nivel inferior en la que se basa la abstracción.
- *Implementador concreto* Clases concretas que implementan la interfaz del implementador. Estas clases proporcionan implementaciones específicas de los métodos definidos por la interfaz del implementador.

2.2. Beneficios del patrón Puente

- **Desacopla la abstracción de la implementación** El principal beneficio del patrón Puente es que divide la capa de abstracción de la capa de implementación. Esto permite que ambas secciones evolucionen de forma independiente, lo que facilita la modificación del código base.
- **Mejora la mantenibilidad** Dado que la base del código está dividida en dos secciones, lo más probable es que realizar cambios en una parte del sistema no afecte a la otra parte. Lo que hace que mantener la base del código sea más fácil y eficiente.
- **Mejora las pruebas** Las pruebas son mucho más fáciles cuando tienes un patrón puente en tu código base porque puedes concentrarte en probar la capa de abstracción por separado de probar la capa de implementación. Esto permite realizar pruebas más fáciles y específicas.
- **Mejora la legibilidad** El patrón Puente crea una jerarquía clara en la base del código. Organizar la base del código de esta manera ayuda a comprender las relaciones entre las diferentes partes del sistema.

2.3. Ejemplo

```
// Abstracción
class Figura {
  constructor(color) {
    this.color = color;
  }

  dibuja() {
    console.log(`Dibujando una Figura con color ${this.color}`);
  }
}

// Implementaciones
class ColorRojo {
  aplicaColor() {
    console.log('Aplicando el color rojo');
  }
}

class ColorAzul {
  aplicaColor() {
    console.log('Aplicando el color azul');
  }
}

// Puente
class FiguraConColor extends Figura {
  constructor(color, implementacionColor) {
    super(color);
    this.implementacionColor = implementacionColor;
  }

  dibuja() {
    super.dibuja();
    this.implementacionColor.aplicaColor();
  }
}

// Uso
const figuraRoja = new FiguraConColor('rojo', new ColorRojo());
const figuraAzul = new FiguraConColor('azul', new ColorAzul());

figuraRoja.dibuja();  // Salida: Dibujando una figura con color rojo, aplicando color rojo
figuraAzul.dibuja(); // Salida: Dibujando una figura con color azul, aplicando color azul
```

3. Composición

El patrón de diseño compuesto permite la creación de objetos con propiedades que son elementos primitivos o una colección de objetos. Imagine una estructura similar a un árbol, donde tiene objetos individuales (nodos de hoja) o grupos de objetos (ramas). El patrón de diseño compuesto le permite crear este tipo de estructura y poder realizar operaciones en cada nivel de manera consistente.

3.1 Componentes del patrón composición

- *Componente* Esta es la interfaz/clase que representa tanto los nodos hoja (elementos individuales) como los nodos compuestos (colección de elementos). El componente define operaciones que se pueden aplicar a ambos tipos de nodos.
- *Hoja* Esto representa objetos individuales en el árbol que no tienen hijos. Implementan las operaciones que se definen en la interfaz del componente.
- *Compuesto* Esto representa los compuestos o contenedores que pueden contener una colección de nodos hoja u otros nodos compuestos.

3.2. Beneficios del patrón composición

- **Uniformidad y consistencia** El patrón de diseño compuesto proporciona una manera uniforme de tratar tanto objetos individuales como composiciones de objetos. Los clientes tienen una interfaz común para operar con estos objetos, lo que simplifica la base del código y las interacciones entre objetos.
- **Flexibilidad** Este patrón de diseño permite flexibilidad para agregar nuevos tipos de componentes o modificar los existentes sin afectar el código del cliente. Puede introducir nuevos tipos de hojas u objetos compuestos fácilmente.
- **Código de cliente simplificado** El código del cliente no necesita distinguir entre componentes individuales y compuestos, lo que hace que trabajar con la estructura sea más sencillo e intuitivo.

3.3. Ejemplo

```
class BloqueUnico {
  constructor(nombre) {
    this.nombre = nombre;
  }

  mostrar() {
    console.log('Bloque:', this.nombre);
  }
}

class ColeccionDeBloques {
  constructor(nombre) {
    this.nombre = nombre;
    this.bloques = [];
  }

  agregar(bloque) {
    this.bloques.push(bloque);
  }

  eliminar(bloque) {
    this.bloques = this.bloques.filter(b => b !== bloque);
  }

  mostrar() {
    console.log('Colección de bloques:', this.nombre);

    for (const bloque of this.bloques) {
      bloque.mostrar();
    }
  }
}

// Usage
const bloque1 = new BloqueUnico('Bloque 1');
const bloque2 = new BloqueUnico('Bloque 2');
const bloque3 = new BloqueUnico('Bloque 3');

const bloqueGrupo1 = new ColeccionDeBloques('Grupo de bloques 1');
bloqueGrupo1.agregar(bloque1);
bloqueGrupo1.agregar(bloque2);

const bloqueGrupo2 = new ColeccionDeBloques('Grupo de bloques 2');
bloqueGrupo2.agregar(bloque3);

const megaEstructura = new ColeccionDeBloques('Megaestructura');
megaEstructura.agregar(bloqueGrupo1);
megaEstructura.agregar(bloqueGrupo2);

megaEstructura.mostrar();
```

4. Decorador

El patrón de diseño Decorador se puede utilizar para modificar el comportamiento de un objeto de forma estática o dinámica sin afectar el comportamiento de otros objetos de la misma clase. Los decoradores son particularmente útiles cuando desea agregar características a un objeto de una manera flexible y reutilizable.

4.1. Componentes del patrón decorador

- *Interfaz de componente* Esto define la lógica de los objetos a los que se les pueden agregar responsabilidades dinámicamente.
- *Componentes concretos* Este es el objeto inicial al que se le pueden agregar funcionalidades adicionales.
- *Decorador* Esta es una interfaz que amplía la funcionalidad de los componentes concretos. Tiene una referencia a una instancia de componente e implementa la interfaz del componente.
- *Decoradores concretos* Estas son las implementaciones concretas de la clase decoradora, agregan un comportamiento específico al componente deseado al extender la clase decoradora.

4.2. Beneficios del patrón decorador

- **Extensibilidad y flexibilidad** Los decoradores le permiten agregar nuevas funcionalidades o comportamientos a los objetos de forma dinámica en tiempo de ejecución. Esto promueve la extensibilidad sin modificar el código base existente y proporciona flexibilidad en cómo se pueden componer y utilizar estas funcionalidades adicionales.
- **Modularidad** Los decoradores permiten un enfoque más modular del código al dividir la funcionalidad en unidades más pequeñas y manejables. Estas unidades se pueden combinar y reutilizar de varias maneras.
- **Configuración de tiempo de ejecución** Los decoradores le permiten configurar dinámicamente un objeto en tiempo de ejecución. Esto le permite agregar o eliminar funcionalidades sin afectar los componentes principales ni tener que volver a compilar el código.
- **Reducir subclases** Sin decoradores, ampliar las funcionalidades suele implicar la creación de numerosas subclases para cada combinación de comportamientos. Los decoradores eliminan la necesidad de subclases, lo que da como resultado un código base más limpio y más fácil de entender.

4.3. Ejemplo

```
class Cafe {
  costo() {
    return 5;
  }
}

class LecheDecorador {
  constructor(cafe) {
    this.cafe = cafe;
  }

  costo() {
    return this.cafe.costo() + 2;
  }
}

class AzucarDecorador {
  constructor(cafe) {
    this.cafe = cafe;
  }

  costo() {
    return this.cafe.costo() + 1;
  }
}

// Uso
let cafe = new Cafe();
console.log('Costo del café simple:', cafe.costo());

let cafeConLeche = new LecheDecorador(cafe);
console.log('Costo del café con leche:', cafeConLeche.costo());

let cafeConLecheAzucarado = new AzucarDecorador(cafeConLeche);
console.log('Costo del café con leche azucarado:', cafeConLecheAzucarado.costo());
```

5. Fachada

El patrón de diseño Facade es básicamente una interfaz simplificada con la que el cliente puede interactuar para utilizar otras operaciones de bajo nivel ocultas en otras partes del código base. Este patrón de diseño se utiliza a menudo en sistemas construidos en torno a una arquitectura multicapa. Las fachadas permiten al cliente realizar determinadas tareas sin necesidad de comprender la complejidad del sistema subyacente.

5.1. Componentes del patrón fachada

- *Fachada*

La fachada es la interfaz con la que interactuará el cliente. Proporciona una interfaz simple y unificada que delega las solicitudes de los clientes a los objetos apropiados dentro del subsistema.

- *Subsistema*

El subsistema consta de todos los diversos componentes y funcionalidades que envuelve la Fachada. El subsistema y la Fachada interactúan entre sí pero operan de forma independiente.

5.2. Beneficios del patrón fachada

- **Interfaz simplificada**

La fachada proporciona una interfaz simple y fácil de entender.

- **Organización del código**

El patrón de diseño Fachada ayuda a organizar el código al encapsular la funcionalidad del subsistema y proporcionar una separación clara de las preocupaciones.

- **Mantenimiento más fácil**

Los cambios en el subsistema se pueden aislar dentro de la fachada, lo que reduce el impacto en el código del cliente.

5.3. Ejemplo

```
// Subsistema de fontanería
class SubsistemaFontaneria {
  constructor() {}

  abrirAgua() {
    console.log('Fontanería: agua abierta');
  }

  cerrarAgua() {
    console.log('Fontanería: Agua cerrada');
  }
}

// Subsistema Eléctrica
class SubsistemaElectrico {
  constructor() {}

  encenderElectricidad() {
    console.log('Eléctrico: Electricidad encendida');
  }

  apagarElectricidad() {
    console.log('Eléctrico: electricidad apagada');
  }
}

// Fachada de la casa
class CasaFachada {
  constructor() {
    this.subsistemaFontaneria = new SubsistemaFontaneria();
    this.subsistemaElectrico = new SubsistemaElectrico();
  }

  entrarEnCasa() {
    this.plumbingSubsystem.abrirAgua();
    this.subsistemaElectrico.encenderElectricidad();
    console.log('Has entrado en la casa.');
  }

  salirDeCasa() {
    this.plumbingSubsystem.cerrarAgua();
    this.subsistemaElectrico.apagarElectricidad();
    console.log('Has salido de casa.');
  }
}

// Cliente
const cliente = () => {
  const casa = new CasaFachada();

  // entrar a la casa
  casa.entrarEnCasa();

  // salir de la casa
  casa.salirDeCasa();
};

// Ejecutamos el cliente
client();
```

6. Objeto ligero

El patrón de diseño Objeto ligero tiene como objetivo minimizar el uso de memoria o los gastos computacionales al almacenar valores intrínsecos (propiedades similares) de objetos similares en una aplicación, reduciendo la cantidad de código duplicado. Esto es particularmente útil cuando se trata de una gran cantidad de objetos similares en una aplicación.

6.1. Componentes de un patrón objeto ligero

- *Factoria de Objetos Ligeros*

La factoría de objetos ligeros crea los objetos ligeros. Contiene una función que creará un objeto ligero si aún no existe uno y almacena objetos ligeros recién creados para futuras solicitudes.

- *Objeto ligero*

El objeto ligero contiene los datos intrínsecos que se compartirán en toda la aplicación.

6.2. Beneficios del patrón objeto ligero

- **Ahorro de memoria**

Al compartir datos intrínsecos entre múltiples objetos, el patrón de Objetos Ligeros reduce significativamente el uso de memoria, especialmente cuando se trata de una gran cantidad de instancias.

- **Mejoras de rendimiento**

Debido al uso reducido de memoria, mejora el rendimiento general del solicitante. Un menor uso de memoria generalmente conduce a tiempos de ejecución más rápidos y un rendimiento más fluido de las aplicaciones.

- **Simplifica la gestión del estado**

Al separar los datos intrínsecos (valores compartidos) y los datos extrínsecos (valores únicos), el patrón de diseño de objeto ligero simplifica la gestión de estos estados. Permite una separación más clara de los cometidos y un enfoque más organizado para el manejo del estado.

6.3. Ejemplo

```
// Objeto ligero para una Camara
function Camara(marca, modelo, resolucion) {
    this.marca = marca;
    this.modelo = modelo;
    this.resolucion = resolucion;
}

// Factoria de objetos ligeros para Camara
var FactoriaObjetoLigeroCamara = (function () {
    var objetosLigeros = {};

    return {
        obtener: function (marca, modelo, resolucion) {
            if (!objetosLigeros[marca + modelo]) {
                objetosLigeros[marca + modelo] = new Camara(marca, modelo, resolucion);
            }
            return objetosLigeros[marca + modelo];
        },

        obtenerContador: function () {
            var contador = 0;
            for (var f in objetosLigeros) contador++;
            return contador;
        }
    };
})();

// Colección de cámaras
function ColeccionCamara() {
    var camaras = {};
    var contador = 0;

    return {
        agregar: function (marca, modelo, resolucion, numeroSerie) {
            camaras[numeroSerie] = {
                flyweight: FactoriaObjetoLigeroCamara.obtener(marca, modelo, resolucion),
                numeroSerie: numeroSerie
            };
            contador++;
        },

        obtener: function (numeroSerie) {
            return camaras[numeroSerie];
        },

        obtenerContador: function () {
            return contador;
        }
    };
}

// Función que ejecutará el ejemplo
function ejecuta() {
    var camaras = new ColeccionCamara();

    camaras.agregar("Canon", "EOS R5", "45MP", "A1234");
    camaras.agregar("Nikon", "D850", "45.7MP", "B5678");
    camaras.agregar("Sony", "A7R III", "42.4MP", "C9101");
    camaras.agregar("Canon", "EOS R5", "45MP", "D1212"); // Reusando el objeto ligero existente

    console.log("Cámaras: " + camaras.obtenerContador());
    console.log("Objetos ligeros: " + FactoriaObjetoLigeroCamara.obtenerContador());
}

// Ejecutamos el ejemplo
ejecuta();
```

7. Apoderado

El patrón de diseño Apoderado es un patrón de diseño estructural que le permite proporcionar un sustituto o marcador de posición para otro objeto. Este objeto apoderado puede controlar el acceso al objeto original. En Javascript, el objeto `Proxy` está integrado en el lenguaje y facilita la implementación del patrón de diseño Apoderado.

7.1. Componentes del patrón Apoderado

- *Apoderado* El Apoderado contiene una interfaz que es similar al objeto real, mantiene una referencia que le permite al apoderado acceder al objeto real y maneja solicitudes y las reenvía al objeto real.
- *Asunto real* Este es el objeto real al que sustituye el apoderado.

7.2. Beneficios del patrón Apoderado

- **Acceso controlado** Los apoderados le permiten controlar el acceso al objeto original, lo que le permite implementar lógica de control de acceso, como permisos, restricciones o validaciones, antes de permitir el acceso al objeto subyacente.
- **Aumento de funcionalidad** Los apoderados pueden agregar comportamiento o funcionalidad adicional antes o después de la invocación de métodos o el acceso a propiedades del objeto original. Esto es útil para implementar cuestiones transversales como el registro, el almacenamiento en caché o el manejo de errores.
- **Almacenamiento en caché**

Los apoderados pueden implementar mecanismos de almacenamiento en caché para almacenar resultados de operaciones costosas, mejorando el rendimiento y la eficiencia.

- **Inicialización diferida**

Los apoderados permiten una inicialización diferida, donde puede retrasar la creación del objeto real hasta que sea necesario. Esto puede mejorar el rendimiento al reducir el uso inicial de recursos.

7.3. Ejemplo

```
// El objeto original representa una cuenta de banco
const cuentaBancaria = {
  saldo: 1000,

  depositar(cantidad) {
    this.saldo += cantidad;
    console.log(`Depositó ${cantidad}. Nuevo saldo: ${this.saldo}`);
  },

  retirar(cantidad) {
    if (cantidad <= this.saldo) {
      this.saldo -= cantidad;
      console.log(`Retiró ${cantidad}. Nuevo saldo: ${this.saldo}`);
    } else {
      console.log('Fondos insuficientes.');
    }
  }
};

// Creamos un proxy para la cuenta bancaria
const bankAccountProxy = new Proxy(cuentaBancaria, {
  // Intercepta el acceso a la propiedad
  obtener(objetivo, propiedad) {
    if (propiedad === 'saldo') {
      // Agrega algún comportamiento personalizado antes de acceder a 'saldo'
      console.log('Saldo accedido.');
    }
    return objetivo[propiedad];
  },

  // Intercepta la invocación al método
  apply(objetivo, thisArg, args) {
    // Agregue algún comportamiento personalizado antes de invocar un método
    console.log(`Se ha llamado al método "${args[0]}".`);
    return objetivo.apply(thisArg, args);
  }
});

// Accedemos al proxy (apoderado)
console.log(bankAccountProxy.saldo); // Esto activará el comportamiento personalizado.

bankAccountProxy.depositar(500); // Esto activará el comportamiento personalizado para la invocación de métodos.
```

Patrones de diseño de comportamiento

Se concentra en cómo los objetos se comunican entre sí y se les asignan las responsabilidades.

1. Cadena de responsabilidad

La Cadena de responsabilidad es un patrón de diseño de comportamiento y su objetivo principal es tomar una solicitud y transmitirla a lo largo de una cadena de manejadores. Cuando la solicitud pasa por la cadena, cada controlador decidirá si procesa la solicitud o la pasa al siguiente controlador de la cadena. Este patrón permite que varios controladores manejen la solicitud sin que el remitente necesite saber cuál la procesará.

1.1. Componentes del patrón Cadena de responsabilidad

- *Solicitud*

La solicitud es solo el objeto que envía el cliente y que debe procesarse. La solicitud pasa por la cadena de controladores hasta que se procesa o llega al final de la cadena.

- *Interfaz/clase del controlador abstracto*

Esta es la interfaz/clase base que define los métodos que se utilizarán para manejar la solicitud. La interfaz del controlador contiene la lógica para el orden de la cadena y cómo se pasa la solicitud.

- *Manipuladores concretos*

Estos son los métodos/clases que implementan el controlador abstracto. Cada controlador concreto contiene lógica para manejar un tipo particular de solicitud. Si el controlador concreto puede procesar la solicitud, lo hará; si no puede, pasará la solicitud al siguiente controlador.

1.2. Beneficios del patrón Cadena de responsabilidad

- **Facilidad de uso**

El remitente no necesita conocer el método específico para procesar la solicitud, solo necesita pasarlo a la cadena. Esto facilita que el remitente procese las solicitudes.

- **Flexibilidad y extensibilidad**

Se pueden agregar o eliminar nuevos controladores a la cadena sin modificar el código del remitente. Esto permite la modificación dinámica del orden de procesamiento.

- **Promueve la segregación responsable**

Los controladores son responsables de procesar tipos específicos de solicitudes según su lógica definida. Esto promueve una clara separación de responsabilidades, lo que facilita la gestión y el mantenimiento de cada manejador de forma independiente.

- **Procesamiento de solicitudes secuenciales**

El patrón garantiza que cada solicitud se procese secuencialmente a través de la cadena de controladores. Cada controlador puede optar por procesar la solicitud o pasarla al siguiente controlador. Esto puede resultar especialmente útil en escenarios en los que las solicitudes deben procesarse en un orden específico.

1.3. Ejemplo

```
// Interfaz del manejador
class CafeManejador {
  constructor() {
    this.siguienteManejador = null;
  }

  establecerSiguiente(manejador) {
    this.siguienteManejador = manejador;
  }

  procesarPeticion(peticion) {
    throw new Error('El método procesarPeticion debe ser implementado por subclases.');
  }
}

// Manejador concreto para pedir un café
class PedirCafeManejador extends CafeManejador {
  procesarPeticion(peticion) {
    if (peticion === 'Cafe') {
      return 'Pedido realizado para una taza de café.';
    } else if (this.siguienteManejador) {
      return this.siguienteManejador.procesarPeticion(peticion);
    } else {
      return 'Lo sentimos, no atendemos ' + peticion + '.';
    }
  }
}

// Manejador concreto para preparar un café
class PrepararCafeManejanador extends CafeManejador {
  procesarPeticion(peticion) {
    if (peticion === 'PrepararCafe') {
      return 'Se está preparando cafe.';
    } else if (this.siguienteManejador) {
      return this.siguienteManejador.procesarPeticion(peticion);
    } else {
      return 'Cannot prepare ' + peticion + '.';
    }
  }
}

// Código del cliente
const pedirManejador = new PedirCafeManejador();
const prepararManejador = new PrepararCafeManejanador();

// Configuramos la cadena de responsabilidad
pedirManejador.establecerSiguiente(prepararManejador);

// Pedimos el café
console.log(pedirManejador.procesarPeticion('Cafe'));  // Salida: Pedido realizado para una taza de café.

// Preparamos el café
console.log(pedirManejador.procesarPeticion('PrepararCafe'));  // Salida: Se está preparando cafe.

// Intenta pedir algo más
console.log(pedirManejador.procesarPeticion('Té'));  // Salida: Lo sentimos, no servimos té.
```

2. Comando

El patrón de diseño de comandos es un patrón de diseño de comportamiento que le permite encapsular una solicitud como un objeto, ese objeto contendrá toda la información necesaria para la ejecución de la solicitud. Este patrón permite la parametrización y puesta en cola de solicitudes y proporciona la capacidad de deshacer operaciones.

2.1. Componentes del patrón Comando

- *Invocador*

Este es el objeto que solicita la ejecución de un comando. Tiene una referencia a un comando y puede ejecutar el comando llamando a su método `ejecutar`. El invocador no necesita conocer los detalles de cómo se ejecuta el comando. Simplemente activa el comando.

- *Comando*

Esta es la interfaz o clase abstracta que declara el método `ejecutar`. Define el método común que las clases de comando concretas deben implementar.

- *Receptor*

Este es un objeto que realiza el trabajo real cuando se llama al método `ejecutar` de un comando. El receptor sabe cómo realizar la acción asociada a un comando específico.

2.2. Beneficios del patrón Comando

- **Flexibilidad y extensibilidad**

Este patrón permite agregar fácilmente nuevos comandos sin necesidad de modificar el invocador o el receptor.

- **Operaciones deshacer y rehacer**

El patrón de comando facilita la implementación de operaciones de deshacer y rehacer. Cada objeto de comando puede realizar un seguimiento del estado anterior, lo que permite revertir la acción ejecutada.

- **Parametrización y colas**

Los comandos se pueden parametrizar con argumentos, lo que permite la personalización de acciones en tiempo de ejecución. Además, el patrón permite poner en cola y programar solicitudes, proporcionando control sobre el orden de ejecución.

- **Historia y registro**

Es posible mantener un historial de comandos ejecutados, lo que puede resultar útil para auditar, registrar o rastrear las acciones del usuario.

2.3. Ejemplo

```
class Comando {
  constructor(receptor) {
    this.receptor = receptor;
  }

  ejecutar() {
    throw new Error('el método ejecutar() debe ser implementado');
  }
}

class ConcretoComando extends Comando {
  constructor(receptor, parametro) {
    super(receptor);
    this.parametro = parametro;
  }

  ejecutar() {
    this.receptor.actuar(this.parametro);
  }
}

class Receptor {
  actuar(parametro) {
    console.log(`El receptor está realizando la acción con parámetro: ${parametro}`);
  }
}

class Invocador {
  constructor() {
    this.comandos = [];
  }

  agregarComando(comando) {
    this.comandos.push(comando);
  }

  ejecutarComandos() {
    this.comandos.forEach(comando => comando.ejecutar());
    this.comandos = []; // Borra los comandos ejecutados
  }
}

// Usage
const receptor = new Receptor();
const comando1 = new ConcretoComando(receptor, 'parámetro del Comando 1');
const comando2 = new ConcretoComando(receptor, 'parámetro del Comando 2');

const invocador = new Invocador();
invocador.agregarComando(comando1);
invocador.agregarComando(comando2);

invocador.ejecutarComandos();
```

3. Intérprete

El patrón de diseño del intérprete se utiliza para definir una gramática para un idioma y proporcionar un intérprete para interpretar oraciones en ese idioma. Por lo general, se usa para crear intérpretes o analizadores de idiomas, pero también puede usarlos dentro de su aplicación. Imagine que tiene una aplicación de escritorio compleja, podría diseñar un lenguaje de programación básico que permita al usuario final manipular su aplicación mediante instrucciones sencillas.

3.1. Componentes del patrón Intérprete

- *Contexto*

Un estado o contexto global que el intérprete utiliza para interpretar expresiones. A menudo contiene información que es relevante durante la interpretación de expresiones.

- *Expresiones abstractas*

Define una interfaz para todo tipo de expresiones en la gramática del idioma. Estas expresiones normalmente se representan como una clase o interfaz abstracta.

- *Expresiones terminales*

Representa los símbolos terminales en la gramática. Estas son las hojas del árbol de expresión. La expresión terminal implementa la interfaz definida por la expresión abstracta.

- *Expresión no terminal*

Representa los símbolos no terminales en la gramática. Las expresiones no terminales utilizan expresiones terminales y/u otras expresiones no terminales para definir expresiones complejas combinándolas o componiéndolas.

- *Árbol de expresiones*

Representa la estructura jerárquica de las expresiones del lenguaje. El árbol de expresiones se construye combinando expresiones terminales y no terminales basadas en las reglas gramaticales del idioma.

- *Intérprete*

Define una interfaz o clase que interpreta el árbol de sintaxis abstracta creado por el árbol de expresión. Por lo general, incluye un método `interpretar` que toma un contexto e interpreta la expresión según el contexto dado.

- *Cliente*

Construye el árbol de sintaxis abstracta (el árbol de expresiones) utilizando expresiones terminales y no terminales basadas en la gramática del idioma. Luego, el cliente utiliza el intérprete para interpretar la expresión.

3.2. Beneficios del patrón Intérprete

- **Facilidad de interpretación de la gramática**

El patrón simplifica la interpretación de reglas gramaticales complejas dividiéndolas en expresiones más pequeñas y manejables. Cada tipo de expresión maneja su propia interpretación, lo que hace que el proceso de interpretación general sea más fácil de gestionar.

- **Mejor manejo de errores**

Dado que cada tipo de expresión maneja su propia interpretación, el manejo de errores se puede adaptar a expresiones específicas. Esto permite mensajes de error más precisos y significativos al analizar o interpretar la entrada.

- **Flexibilidad y extensibilidad**

El patrón de intérprete proporciona una forma flexible de definir y ampliar reglas gramaticales y expresiones del lenguaje sin modificar la lógica central del intérprete. Puede agregar fácilmente nuevas expresiones creando nuevas clases de expresión terminales y no terminales.

- **Integración con otros patrones de diseño**

El patrón Interpreter se puede combinar con otros patrones de diseño, como el patrón de Composición, para crear jerarquías complejas de expresiones. Esto permite la creación de intérpretes potentes y ricos en funciones.

3.3. Ejemplo

```
// Expresión abstracta
class Expresion {
  interpreta(contexto) {
    // Para ser sobreescrito por sus subclases
  }
}

// Expresion terminal: NumeroExpresion
class NumeroExpresion extends Expresion {
  constructor(numero) {
    super();
    this.numero = numero;
  }

  interpreta(contexto) {
    return this.numero;
  }
}

// Expresion terminal: VariableExpresion
class VariableExpresion extends Expresion {
  constructor(variable) {
    super();
    this.variable = variable;
  }

  interpreta(contexto) {
    return contexto[this.variable] || 0;
  }
}

// Expresion no terminal: SumarExpresion
class SumarExpresion extends Expresion {
  constructor(expresion1, expresion2) {
    super();
    this.expresion1 = expresion1;
    this.expresion2 = expresion2;
  }

  interpreta(contexto) {
    return this.expresion1.interpreta(contexto) + this.expresion2.interpreta(contexto);
  }
}

// Expresion no terminal: RestarExpresion
class RestarExpresion extends Expresion {
  constructor(expresion1, expresion2) {
    super();
    this.expresion1 = expresion1;
    this.expresion2 = expresion2;
  }

  interpreta(contexto) {
    return this.expresion1.interpreta(contexto) - this.expresion2.interpreta(contexto);
  }
}

// Código del Cliente
const contexto = { a: 10, b: 5, c: 2 };

const expression = new RestarExpresion(
  new SumarExpresion(
    new VariableExpresion('a'),
    new VariableExpresion('b')
  ),
  new VariableExpresion('c')
);
```

```
const resultado = expression.interpreta(contexto);
console.log('Resultado:', resultado); // Salida: Resultado: 13
```

4. Iterador

El patrón Iterador permite a los clientes recorrer de manera efectiva una colección de objetos.

4.1. Componentes del patrón Iterador

- *Iterador*

Implementa la interfaz o clase Iterador con métodos como `first()`, `next()`. El iterador realiza un seguimiento de la posición actual al atravesar colecciones.

- *Elementos*

Estos son los objetos individuales de la colección que atravesará el iterador.

4.2. Beneficios del patrón Iterador

- **Compatibilidad con diferentes estructuras de datos**

El patrón Iterador permite aplicar la misma lógica de iteración a diferentes estructuras de datos.

- **Soporte para iteración concurrente**

Los iteradores pueden admitir iteraciones simultáneas sobre la misma colección sin interferir entre sí, lo que permite al cliente iterar sobre la colección de diferentes maneras simultáneamente.

- **Carga diferida**

Los iteradores se pueden diseñar para cargar elementos a pedido, lo cual es beneficioso para colecciones grandes donde cargar todos los elementos a la vez puede resultar poco práctico o consumir muchos recursos. Los elementos se pueden recuperar según sea necesario, optimizando el uso de la memoria.

- **Interfaz simplificada**

El patrón Iterador proporciona una interfaz limpia y consistente para acceder a elementos de una colección sin exponer la estructura interna de la colección. Esto simplifica el uso y la comprensión de la colección.

4.3. Ejemplo

```
// clase Coche que representa un coche
class Coche {
  constructor(marca, modelo) {
    this.marca = marca;
    this.modelo = modelo;
  }

  obtenerInformacion() {
    return `${this.marca} ${this.modelo}`;
  }
}

// interfaz Iterador
class Iterador {
  constructor(coleccion) {
    this.coleccion = coleccion;
    this.indice = 0;
  }

  siguiente() {
    return this.coleccion[this.indice++];
  }

  tieneSiguiente() {
    return this.indice < this.coleccion.length;
  }
}

// Colección de coches
class CochesColeccion {
  constructor() {
    this.coches = [];
  }

  agregaCoche(coche) {
    this.coches.push(coche);
  }

  creaIterador() {
    return new Iterador(this.coches);
  }
}

// Uso
const coleccionCoches = new CochesColeccion();
coleccionCoches.agregaCoche(new Coche('Toyota', 'Corolla'));
coleccionCoches.agregaCoche(new Coche('Honda', 'Civic'));
coleccionCoches.agregaCoche(new Coche('Ford', 'Mustang'));

const iterador = coleccionCoches.creaIterador();

while (iterador.tieneSiguiente()) {
  const coche = iterador.siguiente();
  console.log(coche.obtenerInformacion());
}
```

5. Mediador

El patrón Mediador actúa como intermediario entre un grupo de objetos al encapsular cómo interactúan estos objetos. El mediador facilita la comunicación entre los diferentes componentes de un sistema sin necesidad de hacer referencia directa entre sí.

5.1. Componentes del patrón Mediador

- *Mediador*

El mediador gestiona el control central de las operaciones. Contiene una interfaz para comunicarse con los objetos Participante y tiene una referencia a cada objeto Participante.

- *Participante*

Los participantes son los objetos que están siendo mediados, cada participante tiene una referencia al mediador.

5.2. Beneficios del patrón Mediador

- **Control centralizado**

Centralizar la comunicación dentro del Mediador permite un mejor control y coordinación de las interacciones entre los componentes. El Mediador puede gestionar la distribución de mensajes, priorizar mensajes y aplicar lógica específica según los requisitos del sistema.

- **Comunicación simplificada**

Los mediadores simplifican la lógica de la comunicación, ya que los componentes envían mensajes al mediador en lugar de lidiar con la complejidad de la comunicación directa. Esto simplifica la interacción entre componentes y permite un mantenimiento y actualizaciones más fáciles.

- **Reutilizabilidad del mediador**

El Mediador se puede reutilizar en varios componentes y escenarios, lo que permite un único punto de control para diferentes partes de la aplicación. Esta reutilización promueve la coherencia y ayuda a gestionar el flujo de comunicación de manera eficiente.

- **Promueve la mantenibilidad**

El patrón Mediador promueve la mantenibilidad al reducir la complejidad de la comunicación directa entre componentes. A medida que el sistema crece, se pueden realizar cambios y actualizaciones dentro del Mediador sin afectar los componentes individuales, lo que hace que el mantenimiento sea más fácil y menos propenso a errores.

5.3. Ejemplo

```
var Participante = function (nombre) {
    this.nombre = nombre;
    this.salaDeChat = null;
};

Participante.prototype = {
    envia: function (mensaje, para) {
        this.salaDeChat.envia(mensaje, this, para);
    },
    recibe: function (mensaje, de) {
        console.log(de.nombre + " para " + this.nombre + ": " + mensaje);
    }
};

var SalaDeChat = function () {
    var participantes = {};

    return {

        unirse: function (participante) {
            participantes[participante.nombre] = participante;
            participante.salaDeChat = this;
        },

        envia: function (mensaje, de, para) {
            if (para) {                     // mensaje único
                para.recibe(mensaje, de);
            } else {                        // mensaje de difusión
                for (key in participantes) {
                    if (participantes[key] !== de) {
                        participantes[key].recibe(mensaje, de);
                    }
                }
            }
        }
    };
};

function ejecuta() {

    var yoko = new Participante("Yoko");
    var john = new Participante("John");
    var paul = new Participante("Paul");
    var ringo = new Participante("Ringo");

    var salaDeChat = new SalaDeChat();
    salaDeChat.unirse(yoko);
    salaDeChat.unirse(john);
    salaDeChat.unirse(paul);
    salaDeChat.unirse(ringo);

    yoko.envia("Todo lo que necesitas es amor.");
    yoko.envia("Te amo John.");
    john.envia("Oye, no es necesario transmitir", yoko);
    paul.envia("¡Ja, escuché eso!");
    ringo.envia("Paul, ¿qué opinas?", paul);
}
```

6. Recuerdo

El patrón de diseño Recuerdo permite capturar y restaurar el estado de un objeto en un momento posterior sin exponer su estructura interna. El Recuerdo es un objeto separado que almacena el estado del objeto original.

6.1. Componentes del patrón Recuerdo

- *Creador*

Este es el objeto que se guarda. Crea un objeto Recuerdo para almacenar su estado interno.

- *Recuerdo*

El Recuerdo es responsable de almacenar el estado del originador. Tiene métodos para recuperar y establecer el estado del originador pero no expone la estructura interna del originador.

- *Vigilante*

El Vigilante es responsable de realizar un seguimiento y gestionar los Recuerdos. No modifica ni inspecciona el contenido de los Recuerdos.

6.2. Beneficios del patrón Recuerdo

- **Preservación y Restauración del Estado**

Los recuerdos permiten capturar y restaurar el estado interno de un objeto en un momento posterior.

- **Operaciones de deshacer/rehacer**

Recuerdo facilita la implementación de la funcionalidad de deshacer y rehacer fácilmente. Dado que el recuerdo almacena el estado del objeto en varios momentos en el tiempo, puede permitir deshacer los cambios realizados en el objeto o rehacer los cambios realizados en el objeto.

- **Desempeño mejorado**

Almacenar el estado del objeto en un Recuerdo permite operaciones de almacenamiento y recuperación más eficientes en comparación con otros enfoques.

- **Diseño flexible**

Proporciona una forma flexible de gestionar el historial del estado de un objeto. El vigilante puede decidir qué estados conservar y cuándo restaurarlos, lo que permite un enfoque personalizable según los requisitos de la aplicación.

6.3. Ejemplo

```
// Clase Computadora  (Creador)
class Computador {
    constructor() {
      this.so = '';
      this.version = '';
    }

    estableceSO(so, version) {
      this.so = so;
      this.version = version;
    }

    obtenEstado() {
      return {
        so: this.so,
        version: this.version
      };
    }

    restauraEstado(estado) {
      this.so = estado.so;
      this.version = estado.version;
    }
  }

  // Vigilante
  class Vigilante {
    constructor() {
      this.recuerdos = {};
      this.siguienteClave = 1;
    }

    agregar(recuerdo) {
      const clave = this.siguienteClave++;
      this.recuerdos[clave] = recuerdo;
      return clave;
    }

    obtener(clave) {
      return this.recuerdos[clave];
    }
  }

  function ejecuta() {
    const computadora = new Computador();
    const vigilante = new Vigilante();

    // Salvamos el estado
    const estadoOriginal = computadora.obtenEstado();
    const clave = vigilante.agregar(estadoOriginal);

    // Arruinamos el estado
    computadora.estableceSO('Windows', '11');

    // Recuperamos el estado original
    const estadoRestaurado = vigilante.obtener(clave);
    computadora.restauraEstado(estadoRestaurado);

    console.log(computadora.obtenEstado()); // Salida: { so: '', version: '' }
  }

  ejecuta();
```

7. Observador

El patrón de observador permite que muchos objetos se suscriban a eventos transmitidos por otro objeto.

7.1. Componentes del patrón Observador

- *Sujeto*

El sujeto es un objeto que implementa una interfaz que permite a los objetos observadores suscribirse a él y enviar notificaciones a los observadores cuando cambia su estado.

- *Observadores*

El observador se suscribe al sujeto y normalmente tiene una función que se invoca cuando el sujeto lo notifica.

7.2. Beneficios del patrón Observador

- **Manejo de eventos simplificado**

El patrón Observador puede simplificar los mecanismos de manejo de eventos, ya que los eventos pueden tratarse como notificaciones a los observadores sobre un cambio de estado.

- **Reduce el código duplicado**

En lugar de duplicar el mismo código para responder a cambios de estado en varios lugares, el patrón Observador permite un lugar centralizado para gestionar estas respuestas, promoviendo un código más limpio y fácil de mantener.

- **Soporte para comunicación por radiodifusión**

El patrón Observador facilita un modelo de comunicación "uno a muchos", donde un único evento desencadena acciones en múltiples observadores. Esto es útil en escenarios donde varios componentes necesitan reaccionar ante un cambio de estado.

- **Modularidad y reutilización**

Los observadores se pueden reutilizar en diferentes temas, promoviendo la modularidad y la reutilización del código. Esto permite un código más flexible y mantenible.

7.3. Ejemplo

```
function HacerClic() {
    this.manejadores = [];  // observadores
}

HacerClic.prototype = {

    subscribir: function (fn) {
        this.manejadores.push(fn);
    },

    abandonar: function (fn) {
        this.manejadores = this.manejadores.filter(
            function (elemento) {
                if (elemento !== fn) {
                    return elemento;
                }
            }
        );
    },

    disparar: function (o, esteObj) {
        var alcance = esteObj || window;
        this.manejadores.forEach(function (elemento) {
            elemento.call(alcance, o);
        });
    }
}

function ejecutar() {

    var hacerClicManejador = function (elemento) {
        console.log("disparado: " + elemento);
    };

    var hacerClic = new HacerClic();

    hacerClic.subscribir(hacerClicManejador);
    hacerClic.disparar('evento #1');
    hacerClic.abandonar(hacerClicManejador);
    hacerClic.disparar('evento #2');
    hacerClic.subscribir(hacerClicManejador);
    hacerClic.disparar('evento #3');
}

ejecutar();
```

8. Estado

El patrón Estado es un patrón de diseño de comportamiento que le permite tener un objeto base y proporcionarle funcionalidad adicional según su estado. Este patrón es particularmente útil cuando un objeto necesita cambiar su comportamiento en función de su estado interno.

8.1. Componentes del patrón Estado

- *Estado*

Este es el objeto que encapsula los valores del Estado y el comportamiento asociado del Estado.

- *Contexto*

Este es el objeto que mantiene una referencia a un objeto Estado que define el Estado actual. También incluye una interfaz que permite que otros objetos de estado cambien su estado actual a un estado diferente.

8.2. Beneficios del patrón Estado

- **Código modular y organizado**

Cada Estado está encapsulado dentro de su propia clase, lo que hace que el código sea modular y fácil de administrar.

- **No hay necesidad de declaraciones switch**

Las declaraciones switch también se pueden usar para cambiar el comportamiento de un objeto, pero el problema con este método es que las declaraciones switch pueden volverse muy largas a medida que su proyecto escala. El patrón Estado soluciona este problema.

- **Promueve la reutilización**

Los estados se pueden reutilizar en diferentes contextos, lo que reduce la duplicación de código.

- **Simplifica las pruebas**

Probar clases de estados individuales de forma aislada es más fácil y eficaz que probar un objeto monolítico con lógica condicional compleja.

8.3. Ejemplo

```
class Coche {
    constructor() {
      this.estado = new AparcadoEstado();
    }

    establecerEstado(estado) {
      this.estado = estado;
      console.log(`Estado cambiado a : ${estado.constructor.name}`);
    }

    aparcar() {
      this.estado.aparcar(this);
    }

    conducir() {
      this.estado.conducir(this);
    }

    darMarchaAtras() {
      this.estado.darMarchaAtras(this);
    }
  }

  class AparcadoEstado {
    aparcar(coche) {
      console.log("El coche ya está en el estacionamiento");
    }

    conducir(coche) {
      console.log("Cambiando a conducir");
      coche.establecerEstado(new ConduciendoEstado());
    }

    darMarchaAtras(coche) {
      console.log("Cambiando a dar marcha atras");
      coche.establecerEstado(new DandoMarchaAtrasEstado());
    }
  }

  class ConduciendoEstado {
    aparcar(coche) {
      console.log("Cambiando a estacionamiento");
      coche.establecerEstado(new AparcadoEstado());
    }

    conducir(coche) {
      console.log("El coche ya está conduciéndose");
    }

    darMarchaAtras(coche) {
      console.log("Cambiando a dar marcha atras");
      coche.establecerEstado(new DandoMarchaAtrasEstado());
    }
  }

  class DandoMarchaAtrasEstado {
    aparcar(coche) {
      console.log("Cambiando a estacionamiento");
      coche.establecerEstado(new AparcadoEstado());
    }

    conducir(coche) {
      console.log("Cambiando a conducir");
      coche.establecerEstado(new ConduciendoEstado());
    }

    darMarchaAtras(coche) {
      console.log("El coche ya está dando marcha atras");
```

```
    }
  }

  // Ejemplo de uso
  const coche = new Coche();

  coche.conducir();
  coche.darMarchaAtras();
  coche.conducir();
  coche.aparcar();
  coche.conducir();  // Intentando conducir estando estacionado
```

9. Estrategia

El patrón Estrategia es esencialmente un patrón de diseño que le permite tener un grupo de algoritmos (estrategias) que son intercambiables.

9.1. Components del patrón Estrategia

- *Estrategia*

Este es un algoritmo que implementa la interfaz Estrategia.

- *Contexto*

Este es el objeto que mantiene una referencia a la estrategia actual. Define una interfaz que permite al cliente cambiar la estrategia actual a una estrategia diferente o recuperar cálculos de la estrategia actual referenciada.

9.2. Beneficios del patrón Estrategia

- **Algoritmos dinámicamente intercambiables**

Las estrategias se pueden intercambiar en tiempo de ejecución, lo que permite la selección dinámica de algoritmos en función de diferentes condiciones o requisitos. Esto es particularmente útil cuando el algoritmo apropiado puede variar según la entrada del usuario, los ajustes de configuración u otros factores.

- **Flexibilidad y mantenibilidad**

Las estrategias se pueden cambiar o ampliar sin modificar el contexto en el que se utilizan. Esto hace que el sistema sea más flexible y más fácil de mantener, ya que los cambios en una estrategia no afectan a las demás.

- **Simplifica las pruebas**

Probar estrategias de forma aislada es más fácil ya que cada estrategia es una clase separada. Esto permite realizar pruebas específicas y garantiza que los cambios en una estrategia no afecten inadvertidamente a otras.

- **Reutilizabilidad**

Las estrategias se pueden reutilizar en múltiples contextos o aplicaciones, promoviendo la reutilización del código y minimizando la redundancia.

9.3. Ejemplo

```
class ClienteNormalEstrategia {
  calculaPrecio(precioLibro) {
    // Los clientes normales obtienen un descuento fijo del 10%
    return precioLibro * 0.9;
  }
}

class ClienteImportanteEstrategia {
  calculaPrecio(precioLibro) {
    // Los clientes importantes obtienen un descuento fijo del 20%
    return precioLibro * 0.8;
  }
}

class Libreria {
  constructor(preciosEstrategia) {
    this.preciosEstrategia = preciosEstrategia;
  }

  estableceEstrategiaDePrecios(preciosEstrategia) {
    this.preciosEstrategia = preciosEstrategia;
  }

  calculaPrecio(precioLibro) {
    return this.preciosEstrategia.calculaPrecio(precioLibro);
  }
}

// Uso
const clienteNormalEstrategia = new ClienteNormalEstrategia();
const clienteImportanteEstrategia = new ClienteImportanteEstrategia();

const libreria = new Libreria(clienteNormalEstrategia);

console.log('Precio para el cliente normal:', libreria.calculaPrecio(50)); // Salida: 45 (10% de descuento)
libreria.estableceEstrategiaDePrecios(clienteImportanteEstrategia);
console.log('Precio para el cliente importante:', libreria.calculaPrecio(50)); // Salida: 40 (20% de descuento)
```

10. Método Plantilla

El método de plantilla es un patrón de diseño de comportamiento que define el esqueleto del programa de un algoritmo en un método, pero permite que las subclases anulen pasos específicos del algoritmo sin cambiar su estructura.

10.1. Componentes del patrón Método Plantilla

- *Clase abstracta*

La clase abstracta es la plantilla del algoritmo. Define una interfaz para que el cliente invoque su método. También contiene todas las funciones que las subclases pueden sobreescribir.

- *Clase concreta*

Implementa los pasos definidos en la clase abstracta y puede realizar cambios en los pasos.

10.2. Beneficios del patrón Método Plantilla

- **Reutilización del código**

El patrón promueve la reutilización del código al definir el esqueleto del algoritmo en una clase base. Las subclases pueden reutilizar esta estructura y solo necesitan proporcionar implementaciones para pasos específicos.

- **Facil mantenimiento**

Realizar cambios en el algoritmo se simplifica porque las modificaciones solo deben realizarse en un lugar (el método de plantilla en la clase abstracta) en lugar de en varias subclases. Esto reduce las posibilidades de errores y hace que el mantenimiento sea más sencillo.

- **Extensión y variación**

El patrón permite una fácil extensión y variación del algoritmo. Las subclases pueden anular ciertos pasos para proporcionar implementaciones personalizadas, extendiendo o modificando efectivamente el comportamiento del algoritmo sin alterar su estructura central.

- **Flujo de control**

El método de plantilla define el flujo de control del algoritmo, lo que facilita la gestión y comprensión de la secuencia de operaciones del algoritmo.

10.3. Ejemplo

```
cclass Camara {
  // Método de plantilla que define los pasos comunes para capturar una fotografía.
  capturaFoto() {
    this.enciende();
    this.inicializa();
    this.fijaExposicion();
    this.captura();
    this.apaga();
  }

  // Pasos comunes para encender una camara
  enciende() {
    console.log('Encendiendo la cámara');
  }

  // Método abstracto para inicializar la cámara (para ser sobreescrito por subclases)
  inicializa() {
    throw new Error('Método abstracto: inicializa() debe ser implementado por las subclases');
  }

  // Método abstracto para configurar la exposición (para ser anulado por subclases)
  fijaExposicion() {
    throw new Error('Método abstracto: fijaExposicion() debe ser implementado por las subclases');
  }

  // Pasos comunes para capturar una foto
  captura() {
    console.log('Capturando una foto');
  }

  // Pasos comunes para apagar la cámara
  apaga() {
    console.log('Apagando la cámara');
  }
}

class CamaraReflexDigitalObjetivoUnico extends Camara {
  inicializa() {
    console.log('Inicializando la cámara Réflex Digital de Objetivo Único');
  }

  fijaExposicion() {
    console.log('Fijando la exposición de la cámara Réflex Digital de Objetivo Único');
  }
}

class CamaraSinEspejo extends Camara {
  inicializa() {
    console.log('Inicializando la cámara Sin Espejo');
  }

  fijaExposicion() {
    console.log('Fijando la exposición de la cámara Sin Espejo');
  }
}

// Uso
const camaraReflexDigitalObjetivoUnico = new CamaraReflexDigitalObjetivoUnico();
console.log('Capturando foto con la cámara Réflex Digital de Objetivo Único:');
camaraReflexDigitalObjetivoUnico.capturaFoto();
console.log('');

const camaraSinEspejo = new CamaraSinEspejo();
console.log('Capturando foto con la cámara Sin Espejo:');
camaraSinEspejo.capturaFoto();
```

11. Visitante

El patrón de diseño de visitante es un patrón de diseño de comportamiento que le permite separar algoritmos u operaciones del objeto sobre el que operan.

11.1 Componentes del patrón Visitante

- *ObjetoEstructura*

Mantiene una colección de elementos sobre los que se puede iterar.

- *Elementos*

El elemento contiene un método de aceptación que acepta los objetos del visitante.

- *Visitante*

Implementa un método de visita donde el argumento del método es el elemento que se visita. Así es como se realizan los cambios en el elemento.

11.2. Beneficios del patrón Visitante

- **Principio abierto/cerrado**

El patrón se alinea con el Principio Abierto/Cerrado, que establece que las entidades de software (clases, módulos, funciones) deben estar abiertas a la extensión pero cerradas a la modificación. Puede introducir nuevas operaciones (nuevos visitantes) sin modificar la estructura o los elementos del objeto existente.

- **Extensibilidad**

Puede introducir nuevos comportamientos u operaciones agregando nuevas implementaciones de visitantes sin modificar los elementos existentes o la estructura del objeto. Esto hace que el sistema sea más extensible, lo que permite agregar fácilmente nuevas funciones o comportamientos.

- **Comportamiento centralizado**

El patrón Visitante centraliza el código relacionado con el comportamiento dentro de las clases de visitante. Cada visitante encapsula un comportamiento específico, que puede reutilizarse en diferentes elementos, promoviendo la reutilización y la modularidad del código.

- **Consistencia en las operaciones**

Con el patrón Visitante, puede asegurarse de que una operación específica (método de visitante) se aplique de manera consistente en varios elementos, ya que el método de aceptación de cada elemento llama al método de visitante apropiado para ese tipo de elemento.

11.3 Ejemplo

```
class GimnasioMiembro {
    constructor(nombre, tipoSubscripcion, puntuacionFitness) {
        this.nombre = nombre;
        this.tipoSubscripcion = tipoSubscripcion;
        this.puntuacionFitness = puntuacionFitness;
    }

    aceptar(visitante) {
        visitante.visit(this);
    }

    obtenNombre() {
        return this.nombre;
    }

    obtenTipoSubscripcion() {
        return this.tipoSubscripcion;
    }

    obtenPuntuacionFitness() {
        return this.puntuacionFitness;
    }

    establecePuntuacionFitness(puntuacion) {
        this.puntuacionFitness = puntuacion;
    }
}

class EvaluacionFitness {
    visit(miembro) {
        miembro.establecePuntuacionFitness(miembro.obtenPuntuacionFitness() + 10);
    }
}

class DescuentoMembresia {
    visit(miembro) {
        if (miembro.obtenTipoSubscripcion() === 'Premium') {
            console.log(`${miembro.obtenNombre()}: Puntuación Fitness  - ${miembro.obtenPuntuacionFitness()}, Tipo membresía -
        } else {
            console.log(`${miembro.obtenNombre()}: Puntuación Fitness - ${miembro.obtenPuntuacionFitness()}, Tipo membresía -
        }
    }
}

function ejecuta() {
    const miembrosGimnasio = [
        new GimnasioMiembro("Alice", "Basic", 80),
        new GimnasioMiembro("Bob", "Premium", 90),
        new GimnasioMiembro("Eve", "Basic", 85)
    ];

    const evaluacionFitness = new EvaluacionFitness();
    const descuentoMembresia = new DescuentoMembresia();

    for (let i = 0; i < miembrosGimnasio.length; i++) {
        const miembro = miembrosGimnasio[i];

        miembro.aceptar(evaluacionFitness);
        miembro.aceptar(descuentoMembresia);
    }
}

ejecuta();
```

Referencias

Ballard, P. (2018). JavaScript in 24 Hours, Sams Teach Yourself. Sams Publishing.

Crockford, D. (2008). JavaScript: The Good Parts. O'Reilly Media.

Duckett, J. (2011). HTML & CSS: Design and Build Websites. Wiley.

Duckett, J. (2014). JavaScript and JQuery: Interactive Front-End Web Development. Wiley.

Flanagan, D. (2011). JavaScript: The Definitive Guide. O'Reilly Media.

Freeman, E., & Robson, E. (2014). Head First JavaScript Programming: A Brain-Friendly Guide. O'Reilly Media.

Frisbie, M. (2019). Professional JavaScript for Web Developers. Wrox.

Haverbeke, M. (2019). Eloquent JavaScript: A Modern Introduction to Programming. No Starch Press.

Herman, D. (2012). Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript. Addison-Wesley Professional.

McPeak, J., & Wilton, P. (2015). Beginning JavaScript. Wiley.

Morgan, N. (2014). JavaScript for Kids: A Playful Introduction to Programming. No Starch Press.

Murphy C, Clark R, Studholme O, et al. (2014). Beginning HTML5 and CSS3: The Web Evolved. Apress.

Nixon, R. (2021). Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5. O'Reilly Media.

Powell, T., & Schneider, F. (2012). JavaScript: The Complete Reference. McGraw-Hill Education.

Resig, J. (2007). Pro JavaScript Techniques. Apress.

Resig, J., & Bibeault, B. (2016). Secrets of the JavaScript Ninja. Manning Publications.

Robbins, J. N. (2014). Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics. O'Reilly Media.