9. FUNCIONES

Introducción a las funciones: bloques de código reutilizables

Otro concepto esencial en la programación son las funciones, que le permiten almacenar un fragmento de código que realiza una sola tarea dentro de un bloque definido y luego llamar a ese código cada vez que lo necesite con una sola orden corta, en lugar de tener que escribir el mismo código varias veces.

Exploraremos ahora los conceptos fundamentales detrás de las funciones, como:

- la sintaxis básica,
- cómo invocarlas
- cómo definirlas
- el alcance (scope)
- los parámetros.

Junto con los objetos, las funciones son el componente central para comprender JavaScript.

¿Dónde aparecen las funciones?

En JavaScript, encontrarás funciones en todas partes.

- de hecho, hemos estado usando funciones durante todo el curso hasta ahora
- prácticamente cada vez que utilizas una estructura de JavaScript que presenta un par de paréntesis — () — y no está utilizando una estructura de lenguaje integrada común como un bucle for, while o do...while loop, o una sentencia if...`else`, está haciendo uso de una función.

Funciones integradas en el navegador

Hemos utilizado muchas **funciones integradas** (***built-in functions***) en el navegador.

- cada vez que manipulamos una cadena de texto, por ejemplo:

```
const miTexto = 'Soy una cadena';
const nuevaCadena = miTexto.replace('cadena', 'salchicha');
console.log(nuevaCadena);
// la función de cadena replace() toma una cadena fuente,
// y una cadena de destino y reemplaza la cadena de origen,
// con la cadena de destino y devuelve la cadena recién formada
```

- o cada vez que manipulamos un array:

```
const miArray = ['Yo', 'amor', 'chocolate', 'ranas'];
const nuevoString = miArray.join(' ');
console.log(nuevoString);
// la función join() toma un array, se une
// todos los elementos de el array juntos en uno solo
// cadena, y devuelve esta nueva cadena
```

- o cada vez que generamos un número aleatorio:

```
const miNumero = Math.random();
// la función random() genera un número aleatorio entre
// 0 y hasta pero sin incluir 1, y devuelve ese número
```

El lenguaje JavaScript tiene muchas funciones integradas que te permiten hacer cosas útiles sin tener que escribir todo ese código usted mismo.

- de hecho, parte del código que llamas cuando invocas (= ejecutar) una función integrada en el navegador no se puede escribir en JavaScript: muchas de estas funciones usan o invocan partes del **código nativo** del navegador, que está escrito principalmente en lenguajes de sistema de bajo nivel como C++, no en lenguajes web como JavaScript.
- ten en cuenta que algunas funciones integradas del navegador no forman parte del lenguaje principal de JavaScript; algunas se definen como parte de las **API del navegador**, que se basan en el lenguaje predeterminado para proporcionar aún más funcionalidad.

Distinción entre funciones y métodos

Las funciones que forman parte de un **objeto** se denominan **métodos**.

- el código incorporado que hemos utilizado hasta ahora viene en ambas formas: funciones y métodos.

También has visto muchas funciones personalizadas: funciones definidas en tu código, no dentro del navegador. Por ejemplo:

```
function dibujar() {
  ctx.clearRect(0,0,WIDTH,HEIGHT);
  for (let i = 0; i < 100; i++) {
    ctx.beginPath();
    ctx.fillStyle = 'rgba(255,0,0,0.5)';
    ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
    ctx.fill();
  }
}
```

- esta función dibuja 100 círculos aleatorios dentro de un elemento <canvas>.
- cada vez que queramos hacer eso, podemos simplemente **invocar la función** con esto:

```
dibujar();
```

en lugar de tener que escribir todo ese código de nuevo cada vez que queremos repetirlo.

- y las funciones pueden contener cualquier código que desees: incluso puedes llamar a otras funciones desde dentro de funciones.
- la función anterior, por ejemplo, llama tres veces a la función `random()`, que se define mediante el siguiente código:

```
function random(numero) {
  return Math.floor(Math.random() * numero);
}}
```

necesitábamos esta función porque la función `Math.random()` integrada del navegador solo genera un número decimal aleatorio entre 0 y 1: necesitabamos un número entero aleatorio entre 0 y un número específico.

Invocación de/llamada a funciones

Probablemente ya tengas esto claro, pero por si acaso, para usar una función después de haberla definido, debes ejecutarla o invocarla:

- esto se hace incluyendo el nombre de la función en alguna parte del código, seguido de paréntesis.

```
function miFuncion() {
  alert('hola');
}

miFuncion();
// llama a la función una vez
```

Nota: esta forma de crear una función también se conoce como declaración de función.

- siempre está ***hoisted*** (izada/elevada), por lo que puedes llamar a la función desde cualquier parte del script, incluso desde una línea por encima de la definición de función y funcionará bien.

Parámetros de función

Algunas funciones requieren que se especifiquen **parámetros** (= argumentos) cuando las invocas:

- estos son valores que deben incluirse dentro de los paréntesis de la función, que necesita para hacer su trabajo correctamente.

Como ejemplo, la función integrada del navegador `Math.random()` no requiere ningún parámetro. Cuando se llama, siempre devuelve un número aleatorio entre 0 y 1:

```
const miNumero = Math.random();
```

La función integrada del navegador `replace()` necesita dos parámetros:

- la subcadena para buscar en la cadena principal
- la subcadena para reemplazar esa cadena con:

```
const miTexto = 'Soy una cadena';
const nuevaCadena = miTexto.replace('cadena', 'salchicha');
```

Nota: cuando es necesario especificar varios parámetros, se separan con comas.

Parámetros opcionales

A veces, tenemos **parámetros opcionales**, y no es necesario especificarlos.

- si no los especificas, la función generalmente adoptará algún tipo de comportamiento predeterminado.
- como ejemplo, el parámetro de la función array `join()` es opcional
- si no se incluye ningún parámetro para especificar un carácter de unión/delimitación, se utiliza una coma de forma predeterminada.

```
const miArray = ['Me', 'encantan', 'las', 'ranas', 'de', 'chocolate'];
const construirString = miArray.join(' ');
console.log(construirString);
// devuelve 'Me encantan las ranas de chocolate'

const construirOtraCadena = miArray.join();
console.log(hizoOtraCadena);
// devuelve 'I,love,chocolate,frogs'
```

Parámetros predeterminados

Si está escribiendo una función y desea admitir **parámetros opcionales**, puede especificar valores predeterminados agregando `=` después del nombre del parámetro, seguido del **valor predeterminado**:

```
function hola(nombre = 'Chris') {
   console.log(`Hola ${nombre}!`);
}

hola('Ari'); // ¡Hola Ari!
hola();      // ¡Hola Chris!
```

Funciones anónimas y funciones flecha

Hasta ahora acabamos de crear una función como esta:

```
function miFuncion() {
   alerta('hola');
}
```

Pero también puedes crear una función que no tenga nombre:

```
(function () {
   alert('hola');
})
```

- esto se llama una **función anónima**, porque no tiene nombre.
- a menudo verás funciones anónimas cuando una función espera recibir otra función como parámetro.
- en este caso, el parámetro de función a menudo se pasa como una función anónima.

Nota: esta forma de crear una función también se conoce como **expresión función**: a diferencia de la **declaración de función**, las **expresiones función** no se elevan (*hoisted*).

Ejemplo de función anónima

Por ejemplo, supongamos que deseas ejecutar algún código cuando el usuario escribe en un cuadro de texto.

- para hacer esto, puedes llamar a la función `addEventListener()` del cuadro de texto.
- esta función espera que le pases (al menos) dos parámetros:
  - el nombre del evento a escuchar, que en este caso es **`keydown`**
  - una función para ejecutar cuando ocurra el evento.
- cuando el usuario presiona una tecla, el navegador llamará a la función que proporcionó y le pasará un parámetro que contiene información sobre este evento, incluida la tecla particular que presionó el usuario:

```
function logearTecla(evento) {
   console.log(`Presionaste "${evento.key}".`);
}

cajaDeTexto.addEventListener('keydown', logearTecla);
```

En lugar de definir una función `logearTecla()` separada, puedes pasar una función anónima a `addEventListener()`:

```
cajaDeTexto.addEventListener('keydown', function(evento) {
   console.log(`Presionaste "${evento.key}".`);
});
```

Hay otra forma en que las funciones anónimas pueden ser útiles: se pueden **declarar** e **invocar simultáneamente** en una sola expresión, denominada **expresión función invocada inmediatamente** (**IIFE**) :

```
(function () {
// …
})();
```

Un ejemplo de uso de IIFE es emular métodos privados con IIFEs y cierres.

Funciones flecha

Si pasas una función anónima como esta, hay una forma alternativa que puedes usar, llamada **función de flecha**.

- en lugar de `function(evento)`, escribes `(evento) =>` :

```
cajaDeTexto.addEventListener('keydown', (evento) => {
   console.log(`Presionaste "${evento.key}".`);
});
```

Si la función solo tiene una línea entre corchetes, puedes omitir los corchetes:

```
cajaDeTexto.addEventListener('keydown', (evento) => console.log(`Presionaste "${evento.key}".`));
```

Si la función solo toma un parámetro, también puedes omitir los corchetes alrededor del parámetro:

```
cajaDeTexto.addEventListener('keydown', evento => console.log(`Presionaste "${evento.key}".`));
```

Finalmente, si su función necesita devolver un valor y contiene solo una línea, también puede omitir la declaración `return`

- en el siguiente ejemplo, se usa el método `map()` de `Array` para duplicar cada valor en el array original:

```
const originales = [1, 2, 3];
const duplicado = originales.map((elemento) => elemento * 2);

console.log(duplicado); // [2, 4, 6]
```

- el método `map()` toma cada elemento del array a su vez, pasándolo a la función dada.
- luego toma el valor devuelto por esa función y lo agrega a un nuevo array.
- entonces, en el ejemplo anterior, `(elemento) => elemento * 2` es la función de flecha equivalente a:

```
function duplicarElemento(elemento) {
   return elemento * 2;
}
```

Se recomienda usar las funciones de flecha, ya que pueden hacer que tu código sea más corto y más legible.

Ejemplo práctico de función flecha

Aquí hay un ejemplo de trabajo completo del ejemplo "keydown" que discutido anteriormente:

El HTML:

```
<input id="cajaTexto" type="text"></input>
```

```
<div id="salida"></div>
```

Código JavaScript:

```
const cajaTexto = document.querySelector("#cajaTexto");
const salida    = document.querySelector("#salida");

cajaTexto.addEventListener('keydown', (evento) => salida.textContent = `Presionaste "$
{evento.key}".`);
```

Ámbito de funciones y conflictos

Hablemos un poco sobre el **alcance** (***scope***), un concepto muy importante cuando se trata de funciones.

- cuando creas una función, las variables y otras cosas definidas dentro de la función están dentro de su propio alcance separado, lo que significa que están encerradas en sus propios compartimentos separados, inaccesibles desde el código exterior a la función.
- el nivel superior fuera de todas sus funciones se denomina **alcance global** (***global scope***): los valores definidos en el ámbito global son accesibles desde cualquier parte del código.

JavaScript se configura así por varias razones, pero principalmente por motivos de seguridad y organización.

- a veces, no desea que se pueda acceder a las variables desde cualquier parte del código: los scripts externos a los que llama desde otro lugar podrían comenzar a interferir con tu código y causar problemas porque usan los **mismos nombres de variables** que otras partes del código, provocando conflictos.
- esto se puede hacer maliciosamente o simplemente por accidente.

Por ejemplo, supongamos que tienes un archivo HTML que llama a dos archivos JavaScript externos, y ambos tienen una variable y una función definidas que usan el mismo nombre:

```
<!-- Extracto de mi HTML -->
<script src="primero.js"></script>
<script src="segundo.js"></script>
<script>
  saludar();
</script>


// primero.js
const nombre = 'Chris';
function saludar() {
  alert(`Hola ${nombre}: bienvenido a nuestra empresa.`);
}


// segundo.js
const nombre = 'Zaptec';
function saludar() {
  alert(`Nuestra empresa se llama ${nombre}.`);
}
```

Las dos funciones a las que deseas llamar se denominan `saludar()`:

- pero solo puedes acceder a la función `saludar()` del archivo `primero.js` (la segunda se ignora).
- además, se produce un error al intentar (en el archivo `segundo.js`) asignar un nuevo valor a la variable de `nombre`, porque ya se declaró con `const` y, por lo tanto, no se puede reasignar.

Mantener partes de su código encapsuladas en funciones evita tales problemas y se considera la mejor práctica.

Más sobre funciones

Definición de funciones

Declaraciones de funciones

Una **definición de función** (también llamada **declaración de función** o **sentencia `function`**) consta de la **palabra clave `function`**, seguida de:

- el nombre de la función.
- una lista de cero o más **parámetros** (o **argumentos**) de la función, entre paréntesis y separados por comas.
- las declaraciones de JavaScript que definen el cuerpo de la función, encerradas entre corchetes, { /* … */ }.
- un **método** es una función que es una **propiedad de un objeto**
- la setencia **`return`**:
  - se puede usar para devolver un valor en cualquier momento, **finalizando la función** (y **retornando de la función** al punto desde donde se invocó).
  - si no se usa una setencia return (o se usa una sentencia return sin valor), JavaScript devuelve undefined.

Por ejemplo, el siguiente código define una función llamada **`cuadrado`** :

```
function cuadrado(numero) {
   return numero * numero;
}
```

- la función **`cuadrado`** toma un parámetro, llamado **`numero`**.
- la función consta de una **declaración `return`** que indica que devuelve el parámetro de la función (es decir, **`numero`**) multiplicado por sí mismo.
- la **declaración `return`** especifica el valor devuelto por la función:

```
return numero * numero;
```

Las funciones se pueden llamar con más o menos **parámetros** (o **argumentos**) de los que especifica.

- si llama a una función sin pasar los parámetros que espera, se establecerán en undefined.
- si pasa más parámetros de los que espera, la función ignorará los parámetros adicionales.

```
function sumar(x, y) {
  const total = x + y;
  return total;
}

sumar(); // NaN
         // Equivalente a sumar(undefined, undefined)

sumar(2, 3, 4); // 5
                // agregó los dos primeros; 4 fue ignorado
```

- los parámetros se **pasan por valor** a las funciones
  - por lo tanto, si el código dentro del cuerpo de una función asigna un valor completamente nuevo a un parámetro que se pasó a la función, el cambio no se refleja globalmente ni en el código que llamó a esa función.
  - cuando pasa un objeto como parámetro, si la función cambia las propiedades del objeto, ese cambio es visible fuera de la función, como se muestra en el siguiente ejemplo:

```
function miFuncion(elObjeto) {
  elObjeto.marca = 'Toyota';
}

const miCoche = {
  marca : 'Honda',
  modelo: 'Accord',
  anno  : 1998,
};

// a x se le asigna el valor "Honda"
const x = miCoche.marca;

// la función miFuncion cambia la propiedad marca
miFuncion(miCoche);
// se le asigna a y el valor "Toyota"
const y = miCoche.marca ;
```

- cuando pasas un **array** como parámetro, si la función cambia cualquiera de los valores del array, ese cambio es visible fuera de la función, como se muestra en el siguiente ejemplo:

```
function miFuncion(elArray) {
  elArray[0] = 30;
}

const unArray = [45];

console.log(unArray[0]); // 45
miFuncion(unArray);
console.log(unArray[0]); // 30
```

Expresiones función

Si bien la declaración de función anterior es sintácticamente una sentencia, las funciones también se pueden crear mediante una expresión función.

- tal función puede ser **anónima**; no tiene que tener un nombre.
- por ejemplo, la función `cuadrado` podría haberse definido como:

```
const cuadrado = function (numero) {
  return numero * numero;
}
const x = cuadrado(4);  // x obtiene el valor 16
```

Sin embargo, se puede proporcionar opcionalmente un nombre con una **expresión función**, esto:

- permite que la función se llame o invoque a sí misma
- facilita la identificación de la función en los **seguimientos de pila** (***strack trace***) de un **depurador** (***debugger***).

```
const factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
}

console.log(factorial(3))
```

Las expresiones de función son convenientes cuando se **pasa una función como argumento** a otra función.

- el siguiente ejemplo muestra una función de `mapa` que debe recibir una función como primer argumento y un array como segundo argumento:
- en el siguiente código, la función recibe una función definida por una expresión de función y la ejecuta para cada elemento de el array recibido como segundo argumento:

```
function mapear(f, a) {
  const resultado = new Array(a.length);
  for (let i = 0; i < a.length; i++) {
    resultado[i] = f(a[i]);
  }
  return result;
}

const f = function (x) {
  return x * x * x;
}

const numeros = [0, 1, 2, 5, 10];
const cubos = mapear(f, numeros);
console.log(cubos);
```

La función devuelve: [0, 1, 8, 125, 1000].

En JavaScript, una función se puede definir en función de una condición.

- por ejemplo, la siguiente definición de función define `miFuncion` solo si `num` es igual a `0` :

```
let miFuncion;
if (num === 0) {
  miFuncion = function (elObjeto) {
    elObjeto.marca = 'Toyota';
```

```
  }
}
```

Además de definir funciones como se describe aquí, también puede usar el constructor Function para crear funciones a partir de un string en tiempo de ejecución, como hace eval().

Llamada o invocación de funciones

El hecho de declarar o definir una función no la ejecuta.

- definir una función le da un nombre a la función y especifica qué hacer cuando se llama a la función.
- **llamar a la función** en realidad realiza las acciones especificadas en el **cuerpo de la función** con los **parámetros** indicados.
- por ejemplo, si defines la función **`cuadrado`**, podrías llamarla o invocarla de la siguiente manera:

```
cuadrado(5);
```

  - la instrucción anterior llama a la función con un argumento de 5.
  - la función ejecuta sus sentencias y devuelve el valor 25.

Una función debe **estar dentro del alcance** cuando se las llama, pero la **declaración de la función** puede **ser elevada** (***hoisted***)(la definición de la función `f` aparece debajo de la llamada en el código a dicha función `f`).

- el **alcance de la declaración de una función** es la función en la que se declara (o el **programa completo**, si se declara en el nivel superior).

Los **argumentos de una función** no se limitan a cadenas y números: puede pasar objetos de cualquier tipo a una función.

Funciones recursivas

JavaScript le permite llamar a funciones recursivamente.

- esto es particularmente útil para trabajar con estructuras de árbol, como las que se encuentran en el DOM del navegador.
- esto significa que una función puede referirse a sí misma y llamarse a sí misma.
- hay tres formas de que una función se refiera a sí misma:
  - el nombre de la función
  - `arguments.callee`
  - una variable dentro del alcance que hace referencia a la función
- por ejemplo, considere la siguiente **definición de función** :

```
const foo = function bar() {
// declaraciones van aquí
}
```

  - dentro del cuerpo de la función, los siguientes son todos equivalentes:
    - `bar()`
    - `arguments.callee()`
    - `foo()`

Una función que se llama a sí misma se llama función recursiva.

- en cierto modo, la **recursividad** es análoga a un bucle.
- ambos ejecutan el mismo código varias veces y ambos requieren una condición (para evitar un bucle infinito, o más bien, una **recursividad infinita** en este caso).
- por ejemplo, considera el siguiente bucle:

```
let x = 0;
while (x < 10) {       // "x < 10" es la condición del bucle
  //..hacer algo...
  x++;
}
```

  - se puede convertir en una **declaración de función recursiva**, seguida de una llamada a esa función:

```
function bucle(x) {
  if (x >= 10) {    // "x >= 10" es la condición de salida (equivalente a "!(x < 10)")
```

```
    return;
  }
  //..hacer algo...
  bucle(x + 1);     // llamada recursiva
}

bucle(0);
```

- sin embargo, algunos algoritmos no pueden ser bucles iterativos simples.
- por ejemplo, obtener todos los nodos de una estructura de árbol (como el **DOM**) es más fácil a través de la recursividad :

```
function recorrerArbol(nodo) {
  if ( nodo === null) {
    return;
  }
  // do something with node
  for (let i = 0; i < nodo.childNodes.length; i++) {
    recorrerArbol(nodo.childNodes[i]);
  }
}
```

  - en comparación con la función `bucle`, cada **llamada recursiva** en sí misma hace muchas llamadas recursivas a sí misma en este ejemplo anterior.

Es posible convertir cualquier algoritmo recursivo en uno algoritmo no recursivo, pero la lógica suele ser mucho más compleja y hacerlo requiere el uso de una pila.

- de hecho, la recursividad misma usa una pila: la **pila de funciones (*call stack*)**.
- el comportamiento similar a una pila se puede ver en el siguiente ejemplo:

```
function foo(i) {
  if (i < 0) {
    return;
  }
  console.log(`comienzo: ${i}`);
  foo(i - 1);
  console.log(`fin: ${i}`);
}
foo(3);

// Registro:
// comienzo: 3
// comienzo: 2
// comienzo: 1
// comienzo: 0
// fin: 0
// fin: 1
// fin: 2
// fin: 3
```

Otro ejemplo de recursividad:

```
function contarCaracteres(elm) {
  if (elm.tipoDeNodo === 3) {
    // TEXT_NODE
    return elm.nodeValue.length;
  }
  let contador = 0;
  for (let i = 0, hijo; (hijo = elm.childNodes[i]); i++) {
    contador += countChars(hijo);
  }
  return contador;
}
```

Las **expresiones función también** pueden tener nombre, lo que les permite ser recursivas:

```
const caracteresEnElBody = (function contadora(elm) {
  if (elm.tipoDeNodo === 3) {
    // TEXT_NODE
    return elm.nodeValue.length;
```

```
  }
  let contador = 0;
  for (let i = 0, hijo; (hijo = elm.childNodes[i]); i++) {
    contador += contadora(hijo);
  }
  return contador;
})(document.body);
```

- el nombre proporcionado a una expresión función como se indica arriba solo está disponible para el propio alcance de la función.
- esto permite que el motor de JavaScript realice más optimizaciones y da como resultado un código más legible.
- el nombre también aparece en el **depurador** y en algunos **seguimientos de la pila** (***stack traces***), lo que puede ahorrarte tiempo durante la depuración.

Si estás acostumbrado a la **programación funcional**, ten cuidado con las implicaciones de **rendimiento de la recursividad** en JavaScript.

- aunque la especificación del idioma especifica la optimización de la llamada final (tail-call optimization), solo JavaScriptCore (usado por Safari) la ha implementado, debido a la dificultad de recuperar los seguimientos de la pila y la facilidad de depuración.
- si va a existir una **recursividad profunda**, considera usar la **iteración** en lugar de la recursividad para evitar el **desbordamiento de la pila** (***stack overflow***).

Otro ejemplo clásico de función recursiva es el cálculo del factorial:

```
function factorial(n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

- luego podrías calcular los factoriales del `1` al `5` de la siguiente manera:

```
const a = factorial(1); // a obtiene el valor 1
const b = factorial(2); // b obtiene el valor 2
const c = factorial(3); // c obtiene el valor 6
const d = factorial(4); // d obtiene el valor 24
const e = factorial(5); // e obtiene el valor 120
```

Otras formas de invocar funciones

Hay otras formas de llamar funciones: a menudo hay casos en los que

- una función necesita ser llamada dinámicamente,
- el número de argumentos de una función varía,
- el que el contexto de la llamada de función debe establecerse en un objeto específico determinado en tiempo de ejecución.

resulta que cada función es en sí mismo un objeto y, a su vez, como objeto (`Function`) tiene métodos.

- los métodos `call()` y `apply()` se pueden usar sobre una función para invocarla:

```
function Producto(nombre, precio) {
  this.nombre = nombre;
  this.precio = precio;
}

function Comida(nombre, precio) {
  Producto.call(this, nombre, precio);
  this.categoria = 'comida';
}

console.log(new Comida('queso', 5).nombre);
// salida esperada: "queso"
```

Elevación (*hoisting*) de funciones

Considere el siguiente ejemplo:

```
console.log(cuadrado(5)); // 25
```

```
function cuadrado(n) {
   return n * n;
}
```

- este código se ejecuta sin ningún error, a pesar de que se llama a la función `cuadrado()` antes de que se declare.
- esto se debe a que el intérprete de JavaScript **eleva** (***hoists***) la declaración de función completa a la parte superior del alcance actual, por lo que el código anterior es equivalente a:

```
// Todas las declaraciones de funciones están efectivamente en la parte superior del alcance
function cuadrado(n) {
   return n * n;
}

console.log(cuadrado(5)); // 25
```

- la **elevación de funciones** solo funciona con ***declaraciones de* funciones**, no con ***expresiones* función**.
- el siguiente código no funcionará.

```
console.log(cuadrado);
  // ReferenceError : no se puede acceder a 'cuadrado' antes de la inicialización
const cuadrado = function (n) {
  return n * n;
}
```

Alcance de la función (*function scope*)

No se puede acceder a las variables definidas dentro de una función desde ningún lugar fuera de la función, porque la variable se define solo en el ámbito de la función.

- sin embargo, una función puede acceder a todas las variables y funciones definidas dentro del ámbito en el que está definida.
- en otras palabras,
  - una función definida en el **ámbito global** (***global scope***) puede acceder a todas las variables definidas en el ámbito global.
  - una función definida dentro de otra función también puede acceder a todas las variables definidas en su **función principal** y cualquier otra variable a la que tenga acceso la función principal.

```
// Las siguientes variables se definen en el alcance global
const num1 = 20;
const num2 = 3;
const nombre = 'Chamakh';

// Esta función se define en el alcance global
function multiplicar() {
   return num1 * num2 ;
}

multiplicar(); // Devuelve 60

// Un ejemplo de función anidada
function obtenerPuntuacion() {
   const num1 = 2;
   const num2 = 3;

   function sumar() {
      return `${nombre } anotó ${num1 + num2}`;
   }

   return sumar();
}

obtenerPuntuacion(); // Devuelve "Chamakh anotó 5"
```

Funciones anidadas y cierres

Funciones anidadas / funciones internas

Puedes **anidar una función** (***nest***) dentro de otra función.

- la **función anidada** (función **interna**) es privada en su función contenedora (función **externa**).
- también forma un **cierre** (***clousure***).
  - un cierre es una expresión (más comúnmente, una función) que puede tener **variables libres** junto con un entorno que une esas variables (que "cierra" la expresión).
  - dado que una función anidada es un cierre, esto significa que una función anidada puede "heredar" los argumentos y variables de su función contenedora.
  - en otras palabras, la función interna contiene el alcance (scope) de la función externa.
- para resumir:
  - solo se puede acceder a la función interna desde declaraciones en la función externa.
  - la función interna forma un cierre:
    - la función interna puede usar los argumentos y variables de la función externa,
    - mientras que la función externa no puede usar los argumentos y variables de la función interna.
- Esto proporciona una gran utilidad para escribir código más fácil de mantener.
  - si una función llamada se basa en una o dos funciones que no son útiles para ninguna otra parte de su código, puede anidar esas funciones de utilidad dentro de ella.
    - esto mantiene bajo el número de funciones que están en el ámbito global.
  - esto también es un gran atractivo frente al uso de variables globales para lo mismo:
    - al escribir código complejo, a menudo es tentador usar variables globales para compartir valores entre múltiples funciones, lo que lleva a un código que es difícil de mantener.
    - las funciones anidadas pueden compartir variables en su padre, por lo que puede usar ese mecanismo para unir funciones sin contaminar su espacio de nombres global.

El siguiente ejemplo muestra funciones anidadas:

```
function sumarCuadrados(a, b) {
  function cuadrado(x) {
    return x * x;
  }
  return cuadrado(a) + cuadrado(b);
}

const a = sumarCuadrados(2, 3); // devuelve 13
const b = sumarCuadrados(3, 4); // devuelve 25
const c = sumarCuadrados(4, 5); // devuelve 41
```

Dado que la función interna forma un cierre, puede **llamar a la función externa** y especificar argumentos tanto para la función externa como para la interna:

```
function externa(x) {
  function interna(y) {
    return x + y;
  }
  return interna;
}

const funcInterna = externa(3);      // dame una función que suma 3 a cualquier cosa que le pase
const resultado   = funcInterna(5); // devuelve 8
const resultado1  = externa(3)(5);  // devuelve 8
```

Preservación de variables

En el último ejemplo, observa cómo se **conserva** `x` para cada invocación independiente de `interna`

- un cierre debe conservar los **argumentos** y **las variables** en todos los ámbitos a los que hace referencia.
- dado que cada llamada proporciona argumentos potencialmente diferentes, se crea un nuevo cierre para cada llamada al `externa`.
- la memoria asociada a almacenar la información de los argumentos y variables del ámbito, puede liberarse sólo cuando la función interna devuelta ya no es accesible.
- esto no es diferente de almacenar **referencias** en otros objetos, pero a menudo es menos obvio porque uno no establece las referencias directamente y no puede inspeccionarlas.

Funciones anidadas en varios niveles

Las funciones se pueden anidar de forma múltiple.

- por ejemplo:
  - una `funcion(A)` contiene una `funcion(B)`, que a su vez contiene una `funcion(C)`.

- ambas funciones **B** y **C** forman **cierres** aquí, entonces:
  - **B** puede acceder a **A**
  - **C** puede acceder a **B.**
  - además, dado que C puede acceder a B que puede acceder a A, **C** también puede acceder a **A.**

- por lo tanto, los cierres pueden contener **múltiples ámbitos**: contienen recursivamente el alcance de las funciones que lo contienen.
  - esto se llama ***encadenamiento de ámbitos/alcances*** (***scope chaining***).
  - considere el siguiente ejemplo:

```
function A(x) {
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // registra 6 (es decir, 1 + 2 + 3)
```

- en este ejemplo, **C** accede a la y de **B** **y** a la **x** de **A**: esto se puede hacer porque:
  - **B** forma un cierre que incluye a **A** (es decir, **B** puede acceder a los argumentos y variables de **A**).
  - **C forma un cierre que incluye B.**
  - porque el cierre de **C** incluye **B** y el cierre de **B** incluye **A**, entonces el cierre de **C** también incluye **A.**
    - esto significa que **C** puede acceder a los argumentos y variables de tanto **B** *y* **A.**
    - en otras palabras, C encadena los ámbitos de B y A, en ese orden.
  - lo contrario, sin embargo, no es cierto.
    - **A** no puede acceder a **C**, porque **A** no puede acceder a ningún **argumento** o **variable** de **B**, del cual **C** es una variable.
    - por lo tanto, **C** permanece privado solo para **B.**

Conflictos de nombres

Cuando dos argumentos o variables en los ámbitos de un cierre tienen el mismo nombre, hay un conflicto de nombres.

- los ámbitos más anidados tienen prioridad.
- por lo tanto, el **ámbito más interno** tiene la **prioridad más alta**, mientras que el ámbito más externo tiene la más baja.
- esta es la **cadena de ámbito** : el primero de la cadena es el ámbito más interno, y el último es el ámbito más externo.
- considera lo siguiente:

```
function externa() {
  const x = 5;
  function interna(x) {
    return x * 2;
  }
  return interna;
}

externa()(10); // devuelve 20 en lugar de 10
```

- el **conflicto de nombres** ocurre en la declaración **`return x * 2`** y se da entre el parámetro del interior **x** de **`interna`** y de la **variable x** de **`externa`**
- la cadena de alcance aquí es { interna, externa, **objeto global** }.
- por lo tanto, la x de interna tiene prioridad sobre la x de externa, y se devuelve 20 (x de interna) en lugar de 10 (x de externa).

Cierres

Los **cierres** son una de las características más poderosas de JavaScript.

- JavaScript permite **anidar funciones** y otorga a la **función interna** acceso completo a todas las variables y funciones definidas dentro de la **función externa** (y todas las demás variables y funciones a las que tiene acceso la función externa).
- sin embargo, la función externa *no* tiene acceso a las variables y funciones definidas dentro de la función interna.
- esto proporciona una especie de **encapsulación** para las variables de la función interna.
- también, dado que la función interna tiene acceso al alcance/ámbito de la función externa

- las variables y funciones definidas en la función externa vivirán más tiempo que la **duración** de la ejecución de la función externa, si la función interna logra sobrevivir más allá de la vida de la función externa.
- **se crea un cierre cuando la función interna se pone de alguna manera a disposición de cualquier ámbito fuera de la función externa.**

```
const mascota = function (nombre) {    // La función externa define una variable llamada "nombre"
  const obtenerNombre = function () { // La interna tiene acceso a la variable "nombre"
     return nombre;
  }
  return obtenerNombre; // Devuelve la función interna, exponiéndola así a los ámbitos externos
}
const miMascota = mascota('Negrito');

miMascota();              // Devuelve "Negrito"
```

Puede ser mucho más complejo que el código anterior.

- se puede devolver un objeto que contiene métodos para manipular las **variables internas** de la **función externa.**

```
const crearMascota = function (nombre) {
  let sexo;

  const mascota = {
     setNombre(nombreNuevo) {    //  equivale a:  setNombre: function (nombreNuevo)
        nombre = nombreNuevo;
    },

    getNombre() {
      return nombre;
    },

    getSexo() {
      return sexo;
    },

    setSexo(sexoNuevo) {
      if (typeof sexoNuevo === 'string' &&
        (sexoNuevo.toLowerCase() === 'hembra' || sexoNuevo.toLowerCase() === 'macho')) {
        sexo = sexoNuevo;
      }
    }
  };

  return mascota;       // devuelve un objeto que tiene acceso al ámbito de crearMascota
}

const mascota = crearMascota('Linda');
mascota.getNombre();                    // Linda
mascota.setNombre('Toxo');
mascota.setSexo('macho');
mascota.getSexo();                      // macho
mascota.getNombre();                    // Toxo
```

- en el código anterior, la variable de **nombre** de la función externa es accesible para las funciones internas, y no hay otra forma de acceder a las variables internas excepto a través de las funciones internas.
- las variables internas de las funciones internas actúan como almacenes seguros para los argumentos y variables externos.
- contienen datos "persistentes" y "encapsulados" para que trabajen las funciones internas.
- las funciones ni siquiera tienen que estar asignadas a una variable, o tener un nombre.

Otro ejemplo de uso de cierres:

```
const getCodigo = (function () {
  const codigoAPI = '0]Eal(eh&2';  // Código que no deseamos que se pueda modificar externamente…

  return function () {
    return codigoAPI;
  };
})();

getCodigo();    // Devuelve el  codigoAPI
```

Nota: hay una serie de precauciones que hay que tener en cuenta al usar cierres

- si una función encerrada define una variable con el mismo nombre que una variable en el ámbito externo, entonces no hay forma de volver a referirse a la variable en el ámbito externo.
- la variable del ámbito interno " anula " la externa, hasta que el programa sale del ámbito interno. Se puede considerar como **un conflicto de nombres**.

Ejemplo:

```
const crearMascota = function ( nombre ) {
   return {
      setNombre( nombre ) { // La función interna también define una variable llamada "nombre".
         nombre = nombre ; // ¿Cómo accedemos al "nombre" definido por la función externa?
      }
   }
}
```

Como usar el objeto `arguments`

Los **argumentos de una función** se mantienen en un objeto similar a un array.

- dentro de una función, puedes acceder a los argumentos que se le pasan de la siguiente manera:

```
arguments[i]
```

- donde i es el número ordinal del argumento, comenzando en 0.
- entonces, el primer argumento pasado a una función sería arguments[0].
- el número total de argumentos se indica mediante **`argumentos.length`**
- el invocador de la función se indica mediante **`argumentos.callee`**

Mediante el uso del objeto de **`arguments`**

- puedes **llamar a una función** con más argumentos de los que se **declara formalmente** para aceptar.
  - esto suele ser útil si no sabe de antemano cuántos argumentos se pasarán a la función.
  - puedes usar **`argumentos.length`** para determinar el número de argumentos realmente pasados a la función y luego acceder a cada argumento usando el objeto **`arguments`**.
- por ejemplo, considera una función que concatene varias cadenas.
  - el único **argumento formal** para la función es una cadena que especifica los caracteres que separan los elementos a concatenar.
  - la función se define de la siguiente manera:

```
function miConcat(separador) {
  let resultado = '';                       // inicializar lista

  for (let i = 1; i < arguments.length; i++) {    // iterar por todos los argumentos
    resultado += arguments[i] + separador;
  }
  return resultado;
}
```

    - puede pasar cualquier cantidad de argumentos a esta función, y concatena cada argumento en una "lista" de cadena:

```
// devuelve "rojo, naranja, azul, "
miConcat(', ', 'rojo', 'naranja', 'azul');

// devuelve "elefante; jirafa; león; guepardo; "
miConcat('; ', 'elefante', 'jirafa', 'león', 'guepardo');

// devuelve "salvia. albahaca. orégano. pimienta. perejil. "
miConcat('. ', 'salvia', 'albahaca', 'orégano', 'pimienta', 'perejil');
```

Nota: la variable arguments es "parecida a un array", pero no un array.

- es similar a un array en el sentido de que tiene un índice numerado y una propiedad length.
- sin embargo, no posee todos los métodos de manipulación de arrays.

Parámetros de función

Hay dos tipos especiales de **sintaxis de parámetros** :

- **parámetros predeterminados** (***default parameters***).
- **parámetros restantes** (***rest parameters***).

Parámetros predeterminados

En JavaScript, los parámetros de las funciones por defecto son `undefined`

- sin embargo, en algunas situaciones, puede ser útil establecer un valor predeterminado diferente: esto es exactamente lo que hacen los parámetros predeterminados.
- en el pasado, la estrategia general para establecer valores predeterminados era probar los valores de los parámetros en el cuerpo de la función y asignar un valor si no estaban definidos.
- en el siguiente ejemplo, si no se proporciona ningún valor para b, su valor no estaría definido al evaluar a*b, y una llamada a multiplicar normalmente habría devuelto NaN.
- sin embargo, esto se evita con la segunda línea de este ejemplo:

```
function multiplicar(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;
  return a * b;
}

multiplicar(5); // 5
```

- con parámetros predeterminados, ya no es necesaria una verificación manual en el cuerpo de la función.
- puede poner `1` como valor predeterminado para `b` en el encabezado de la función:

```
function multiplicar(a, b = 1) {
  return a * b;
}

multiplicar(5); // 5
```

Parámetros con nombre

JavaScript no tiene parámetros con nombre, sin embargo es posible implementarlos utilizando la desestructuración de objetos, que permite empaquetar y desempaquetar objetos de manera conveniente.

```
// Tenga en cuenta las llaves { }: esto es desestructurar un objeto
function area( { ancho, alto } ) {
   return ancho * alto;
}

// Las llaves { } aquí crean un nuevo objeto
console.log(area( { ancho: 2, alto: 3 } ));
```

Parámetros...rest*antes*

La sintaxis de **parámetros rest** nos permite representar un número indefinido de argumentos como un array.

- en el siguiente ejemplo, la función `multiplicar` utiliza ***parámetros rest / parámetros restantes*** para recopilar argumentos desde el segundo hasta el final.
- la función luego los multiplica por el primer argumento.

```
function multiplicar(multiplicador,...losArgumentos) {
  return losArgumentos.map( (x) => multiplicador * x);
}

const arr = multiplicar(2, 1, 2, 3);
console.log(arr);                       // [2, 4, 6]
```

Si una función acepta una lista de argumentos y ya tienes un array, puedse usar la **sintáxis spread** dentro de la llamada a la función

para **desplegar el array** (o transformar el array) en una lista de elementos:

- por ejemplo, supongamos que tenemos la función:

```
function calcularMedia(...argumentos) {
  let suma = 0;
  for (const elemento of argumentos) {
    suma += elemento;
  }
  return suma / argumentos.length;
}

calcularMedia(2, 3, 4, 5); // 3.5
```

- entonces, si podemos usar la sintaxis spread así:

```
let numeros = [2, 3, 4, 5];
calcularMedia(...numeros);
```

Funciones flecha

Una expresión **función flecha** (también llamada ***flecha gruesa*** para distinguirla de una hipotética sintaxis **->** en futuro JavaScript)

- tiene una sintaxis más corta en comparación con las **expresiones función**
- no tiene sus propios argumentos this, super o new.target.
- las funciones de flecha son siempre anónimas.

Dos factores influyeron en la introducción de las funciones de flecha:

- funciones más cortas
- poder usar this dentro de una función sin crear una ligadura (es decir, un objeto asocidado).
- disponer de otra manera de crear funciones anónimas

new.target

La **metapropiedad** `new.target` te permite detectar si una función o constructor fue llamado usando el operador `new`:

- en constructores y funciones invocados usando el operador **new**, `new.target` devuelve una referencia al constructor o función a la que se llamó **new**.
- en las llamadas a funciones normales, `new.target` es `undefined`.

Ejemplo de código:

```
function Foo() {
  if (!new.target) { throw 'Foo() se debe invocar a través de new'; }
}

try {
  Foo();
} catch (e) {
  console.log(e);
  // salida esperada: "Foo() se debe invocar a través de new"
}
```

Funciones más cortas

En algunos patrones funcionales, las funciones más cortas son bienvenidas.

- comparar:

```
constante a = [
'Hidrógeno',
'Helio',
'Litio',
'Berilio'
];

const a2 = a.map( function(s) { return s.length; } );
console.log(a2); // [9, 5, 5, 7]

const a3 = a.map( (s) => s.length );
```

```
console.log(a3); // [9, 5, 5, 7]
```

this no ligado

Hasta la aparición de las funciones flecha

- cada nueva función tiene su propio valor `this` es decir:
  - un nuevo objeto en el caso de un constructor,
  - `undefined` en llamadas de función de modo estricto,
  - el objeto base / objeto global si la función se llama como un "método de objeto",
  - etc.
- esto resultó ser un engorro con un estilo de programación orientado a objetos.

```
function Persona() {
  this.edad = 0;        // el constructor Persona() define `this` como sí mismo.

  setIntervalo(
     function crecer() { this.edad++;},      //  En modo no-estricto, la función crecer define
     1000                                    //  'this' como el objeto global, lo cual es diferente
  );                                         //  del 'this' del constructor Persona
}

const p = new Persona();
```

- En ECMAScript 3/5, este problema se solucionó asignando el valor de `this` a una variable que se podía cerrar.

```
function Persona() {
  const yoMismo = this; // Algunos escogen 'that' o 'self' en lugar de 'yoMismo'
                        // La cuestión es elegir uno y ser coherente en su uso.
  yoMismo.edad = 0;

  setIntervalo(
       function crecer() {  yoMismo.edad++; },
       1000
  );
}
```

- alternativamente, se puede crear una función enlazada para que el valor apropiado de this se pase a la función crecer().

Una función de flecha no tiene su propio `this`; se utiliza el valor `this` del contexto de ejecución adjunto.

- por lo tanto, en el siguiente código, `this` dentro de la función que se pasa a `setIntervalo` tiene el mismo valor que `this` en la función adjunta :

```
function Persona() {
  this.edad = 0;

  setInterval( () => {
    this.edad++;          // `this` en este contexto se refiere al objeto Persona
  }, 1000);
}

const p = new Persona();
```

Funciones anónimas flecha

Hay otra forma de definir funciones anónimas: usando una expresión de función flecha.

```
// Observa que no hay un nombre de función antes de los paréntesis
const calcularMedia = (...argumentos) => {
  let suma = 0;
  for (const elemento of argumentos) {
    suma += elemento;
  }
  return suma / argumentos.length;
};
```

```
// Puedes omitir return cuando la función flecha consta de una única expresión
const sumar = (a, b, c) => a + b + c;
```

Las funciones flecha no son semánticamente equivalentes a las expresiones función.

Funciones predefinidas

JavaScript tiene varias **funciones integradas de alto nivel (*built-in functions*)**:

- `eval()`: evalúa el código JavaScript representado como una cadena.
- `isFinite()`: determina si el valor pasado es un número finito.
  - si es necesario, el parámetro se convierte primero en un número.
- `isNaN()`: determina si un valor es `NaN` o no.
  - Nota: la coerción dentro de la función `isNaN` tiene reglas interesantes; alternativamente, puede querer usar `Number.isNaN()` para determinar si el valor es Not-A-Number.
- `parseFloat()`: analiza un argumento de cadena y devuelve un número de punto flotante.
- `parseInt()`: analiza un argumento de cadena y devuelve un número entero de la base especificada (la base en los sistemas numéricos matemáticos).
- `decodeURI()`: decodifica un **identificador uniforme de recursos** (**URI**) creado previamente por `encodeURI` o por una rutina similar.
- `decodeURIComponent()`: decodifica un componente de identificador uniforme de recursos (URI) creado previamente por `encodeURIComponent` o por una rutina similar.
- `encodeURI()`: codifica un identificador uniforme de recursos (URI) reemplazando cada instancia de ciertos caracteres por una, dos, tres o cuatro secuencias de escape que representan la codificación UTF-8 del carácter (solo habrá cuatro secuencias de escape para caracteres compuestos por dos caracteres "sustitutos").
- `encodeURIComponent()`: codifica un componente de identificador uniforme de recursos (URI) reemplazando cada instancia de ciertos caracteres por una, dos, tres o cuatro secuencias de escape que representan la codificación UTF-8 del carácter (solo habrá cuatro secuencias de escape). secuencias para caracteres compuestos por dos caracteres "sustitutos").
- `escape()`: (en desuso) calcula una nueva cadena en la que ciertos caracteres han sido reemplazados por una secuencia de escape hexadecimal. Utiliza **encodeURI** o **encodeURIComponent en su** lugar.
- **`unescape(): (en desuso) calcula una nueva cadena en la que las secuencias de escape hexadecimales se reemplazan con el carácter que representa. Las secuencias de escape pueden ser introducidas por una función como escape.`** Debido a que `unescape()` está en desuso, use `decodeURI()` o `decodeURIComponent` **en su** lugar.

Más sobre Cierres

Un **cierre** (***clousure***)

- es la combinación de "empaquetar" una función y un conjunto de referencias a su estado circundante (el **entorno léxico**).
  - un cierre te da acceso al alcance de una función externa desde una función interna.
- en JavaScript, los cierres se crean cada vez que se crea una función, en el momento de la creación de la función.

Alcance léxico

Considere el siguiente código de ejemplo:

```
function inicializar() {
  var nombre = 'Mozilla';         // nombre: es una variable local creada por inicializar
  function mostrarNombre() {      // mostrarNombre(): es la función interna, un cierre
    console.log(nombre);          // usa la variable declarada en la función padre
  }
  mostrarNombre();
}
inicializar();
```

- `inicializar()` crea una variable local llamada **`nombre`** y una función llamada `mostrarNombre()`.
- la función mostrarNombre() es una **función interna** que se define dentro de `inicializar()` y está disponible solo dentro del cuerpo de la función **`inicializar()`**.
- la función `mostrarNombre()` no tiene variables locales propias.
- sin embargo, dado que las funciones internas tienen acceso a las variables de las funciones externas, **`mostrarNombre()`** puede acceder al nombre de la variable declarada en la función principal, **`inicializar()`**.

Si ejecutamos el código anterior observamos que:

- la instrucción **`console.log()`** dentro de la función **`mostrarNombre()`** muestra correctamente el valor de la variable de **`nombre`**, que se declara en su función principal.
- este es un ejemplo de **alcance léxico**, que describe cómo un el **analizador sintáctico** (***parser***) resuelve nombres de variables cuando las funciones están anidadas.
- la palabra ***léxico*** se refiere al hecho de que el alcance léxico usa la ubicación donde se declara una variable dentro del código fuente para determinar dónde está disponible esa variable.
- las **funciones anidadas** tienen acceso a las variables declaradas en su ámbito externo.

En este ejemplo en particular, el alcance se denomina ***alcance de función***, porque la variable es accesible y solo se puede acceder dentro del cuerpo de la función donde se declara.

Alcance con `let` y `const`

Tradicionalmente (antes de ES6), JavaScript solo tenía dos tipos de ámbitos: ámbito de función y ámbito global.

- las variables declaradas con `var` tienen alcance de función o alcance global, dependiendo de si se declaran dentro o fuera de una función.
- esto puede ser lioso, porque los bloques con llaves no crean ámbitos:

```
if (Math.random() > 0.5) {
  var x = 1;
} else {
  var x = 2;
}
console.log(x);
```

- para personas de otros lenguajes (por ejemplo, C, Java) donde los bloques crean ámbitos, el código anterior debería generar un error en la línea `console.log`, porque estamos fuera del ámbito de x en cualquiera de los bloques.
- sin embargo, debido a que los bloques no crean ámbitos para `var`, las instrucciones `var` aquí en realidad crean una variable global.
- también hay un ejemplo práctico presentado a continuación que ilustra cómo esto puede causar errores reales cuando se combina con cierres.

En ES6, JavaScript introdujo las declaraciones `let` y `const`, que, entre otras cosas, como **zonas muertas temporales** (`TDZ`, ***Temporal Dead Zones***), te permiten crear variables de ámbito de bloque.

```
if (Math.random() > 0.5) {
```

```
  const x = 1;
} else {
  const x = 2;
}

console.log(x); // ReferenceError: x no está definida
```

En esencia

- los bloques finalmente se tratan como ámbitos en ES6, pero solo si declara variables con `let` o `const`.
- además, ES6 introdujo **módulos**, que introdujeron otro tipo de alcance.
- los cierres son capaces de capturar variables en todos estos ámbitos, como se verá en breve.

Cierre

Considera el siguiente ejemplo de código:

```
function funcionHacer() {
  const nombre = 'Mozilla';

  function mostrarNombre() {
    console.log(nombre);
  }

  return mostrarNombre;
}

const miFuncion = funcionHacer();
miFuncion();
```

- ejecutar este código tiene exactamente el mismo efecto que el ejemplo anterior de la función `inicializar()` anterior.
- lo que es diferente (e interesante) es que **`mostrarNombre()`**, la función interna se devuelve desde la función externa antes de ejecutarse.
- a primera vista, puede parecer poco intuitivo que este código tambien funcione:
  - en algunos lenguajes de programación, las variables locales dentro de una función existen solo durante la ejecución de esa función.
  - una vez que `funcionHacer()` termina de ejecutarse, es de esperar que la variable de nombre ya no sea accesible.
  - sin embargo, debido a que el código aún funciona como se esperaba, obviamente este no es el caso en JavaScript.
- la razón es que las funciones en JavaScript forman cierres.
  - un ***cierre*** es la combinación de una **función** y el **entorno léxico** dentro del cual se declaró esa función.
  - este entorno consta de cualquier **variable local** que estuviera dentro del alcance en el momento en que se creó el cierre.
  - en este caso, `miFuncion` es una referencia a la instancia de la función `mostrarNombre` que se crea cuando se ejecuta `funcionHacer`.
  - la instancia de `mostrarNombre` mantiene una referencia a su entorno léxico, dentro del cual existe el nombre de la variable.
  - por este motivo, cuando se invoca `miFuncion`, el nombre de la variable permanece disponible para su uso y "Mozilla" se pasa a `console.log`.

Aquí hay un ejemplo un poco más interesante: una función `hacerSumardor` :

```
function hacerSumardor(x) {
  return function (y) {
    return x + y;
  };
}

const sumar5  = hacerSumardor(5);
const sumar10 = hacerSumardor(10);

console.log(sumar5(2));     // 7
console.log(sumar10(2));    // 12
```

- en este ejemplo, hemos definido una función `hacerSumardor(x)`, que toma un único argumento x y devuelve una nueva función.
- la función que devuelve toma un solo argumento y, y devuelve la suma de x e y.
- en esencia, `hacerSumardor` es una **fábrica de funciones** (**function factory**).
  - crea funciones que pueden agregar un valor específico a su argumento.

    - en el ejemplo anterior, la fábrica de funciones crea dos nuevas funciones: una que agrega cinco a su argumento y otra que agrega 10.
- `sumar5` y `sumar10` son ambos cierres.
    - comparten la misma definición de cuerpo de función, pero almacenan diferentes entornos léxicos.
    - en el entorno léxico de `sumar5`, **`x`** es 5, mientras que en el entorno léxico de `sumar10`, **`x`** es 10.

Cierres prácticos

Los cierres

- son útiles porque te permiten asociar datos (el entorno léxico) con una función que opera sobre esos datos.
- esto tiene paralelos obvios con la programación orientada a objetos, donde los objetos te permiten asociar datos (las propiedades del objeto) con uno o más métodos.

En consecuencia, puedes usar un cierre en cualquier lugar donde normalmente usarías un objeto con un solo método.

- las situaciones en las que es posible que desee hacer esto son particularmente comunes en la web.
- gran parte del código escrito en el front-end JavaScript está basado en eventos.
    - defines algún **comportamiento** y luego lo adjuntas a un **evento** que desencadena el usuario (como un clic o una pulsación de tecla).
    - el código se adjunta como una **devolución de llamada** (**callback)** (una sola función que se ejecuta en respuesta al evento).

Por ejemplo, supongamos que queremos agregar botones a una página para ajustar el tamaño del texto.

- una forma de hacer esto es especificar el tamaño de fuente del elemento del cuerpo (en píxeles) y luego establecer el tamaño de los otros elementos en la página (como los encabezados) usando la **unidad em relativa**
- dichos botones interactivos de tamaño de texto pueden cambiar la propiedad de tamaño de fuente del elemento del cuerpo, y los ajustes son recogidos por otros elementos en la página gracias a las unidades relativas.

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}

h2 {
  font-size: 1.2em;
}
```

Aquí está el JavaScript:

```
function hacerRedimensionador(tamano) {
  return function () {
    document.body.style.fontSize = `${tamano}px`;
  };
}

const size12 = hacerRedimensionador(12);
const size14 = hacerRedimensionador(14);
const size16 = hacerRedimensionador(16);
```

- **`size12`**, **`size14`** y **`size16`** ahora son funciones que cambian el tamaño del cuerpo del texto a 12, 14 y 16 píxeles, respectivamente.
- puedes adjuntarlos a botones (en este caso, hipervínculos) como se muestra en el siguiente ejemplo de código.

```
document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;

<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-16">16</a>
```

Emulación de métodos privados con cierres

Los lenguajes como Java le permiten declarar métodos como privados, lo que significa que solo pueden ser llamados por otros métodos en la misma clase.

- JavaScript, antes de las clases, no tenía una forma nativa de declarar métodos privados, pero era posible emular métodos privados usando cierres.
- los métodos privados no solo son útiles para restringir el acceso al código: también brindan una forma poderosa de administrar su **espacio de nombres** global.

El siguiente código ilustra cómo usar los cierres para definir funciones públicas que pueden acceder a **funciones privadas** y **variables privadas**.

- se dice que estos cierres siguen el **Patrón de Diseño Modular** (***Module Design Pattern***).

```
const contador = (function () {
  let contadorPrivado = 0;

  function cambiarPor(valor) {
    contadorPrivado += valor;
  }

  return {
    incrementar() { cambiarPor(1); },
    decrementar() { cambiarPor(-1);},
    valor()       { return contadorPrivado; },
  };
})();

console.log(contador.valor()); // 0.

contador.incrementar();
contador.incrementar();

console.log(contador.valor()); // 2.

contador.decrementar();
console.log(contador.valor()); // 1.
```

En el ejemplo anterior, cada cierre tenía su propio entorno léxico:

- aquí, sin embargo, hay un único entorno léxico que comparten las tres funciones:
  - `contador.incrementar`
  - `contador.decrementar`
  - `contador.valor`.
- el **entorno léxico** compartido se crea en el cuerpo de una función anónima, *que se ejecuta tan pronto como se define* (también conocida como **IIFE** = **Expresión función invocada inmediatamente**).
- el entorno léxico contiene dos elementos privados:
  - una variable llamada `contadorPrivado` y una función llamada `cambiarPor`.
  - no puede acceder a ninguno de estos miembros privados desde fuera de la función anónima.
  - en su lugar, puede acceder a ellos utilizando las tres funciones públicas que se devuelven desde el **contenedor anónimo**.
  - esas tres funciones públicas son **cierres** que comparten el mismo entorno léxico.
  - gracias al alcance léxico de JavaScript, cada uno tiene acceso a la variable `contadorPrivado` y la función `cambiarPor`.

En el siguiente ejemplo, continuación del anterior:

- observe cómo los dos *contadores* mantienen su independencia entre sí.
- cada cierre hace referencia a una versión diferente de la variable `contadorPrivado` a través de su propio cierre.
  - cada vez que se llama a uno de los contadores, su **entorno léxico** cambia cambiando el valor de esta variable.
  - los cambios en el valor de la variable en un cierre no afectan el valor en el otro cierre.

```
const hacerContador = function () {
  let contadorPrivado = 0;

  function cambiarPor(valor) {
    contadorPrivado += valor;
  }

  return {
```

```
    incrementar() { cambiarPor(1); },
    decrementar() { cambiarPor(-1);},
    valor()       { return contadorPrivado; },
  };
};


const contador1 = hacerContador();
const contador2 = hacerContador();

console.log(contador1.valor());     // 0
console.log(contador2.valor());     // 0

contador1.incrementar();
contador1.incrementar();
console.log(contador1.valor());     // 2

contador1.decrementar();
console.log(contador1.valor());     // 1
console.log(contador2.valor());     // 0
```

Nota: el uso de cierres de esta manera proporciona beneficios que normalmente se asocian con la programación orientada a objetos: en particular, la **ocultación** y **encapsulación de datos**.

Cadena de alcance de cierre

Todo cierre tiene tres alcances:

- **Alcance local** (**Alcance propio**)
- **Alcance envolvente** (puede ser alcance de bloque, alcance de función o alcance de módulo)
- **Alcance global**

Un error común es no darse cuenta de que en el caso de que la función externa sea una función anidada, el acceso al alcance de la función externa incluye el alcance envolvente de la función externa: creando en la práctica una **cadena de alcances de función** (***chain of function scopes***).

Para entenderlo mejor, considera el siguiente código de ejemplo.

```
// Esto es ámbito global
const e = 10;
function sumar(a) {
  return function (b) {
    return function (c) {
      // ámbito de las funciones más externas
      return function (d) {
        // ámbito local de esta función
        return a + b + c + d + e;
      };
    };
  };
}

console.log(sumar(1)(2)(3)(4)); // 20
```

También puede escribir sin **funciones anónimas** :

```
// Esto es ámbito global
const e = 10;
function sumar(a) {
  return function sumarDos(b) {
    return function sumarTres(c) {
      // ámbito de las funciones más externas
      return function sumarCuatro(d) {
        // ámbito local de esta función
        return a + b + c + d + e;
      };
    };
  };
}
```

```
const sumar2 = sumar(1);
const sumar3 = sumar2(2);
const sumar4 = sumar3(3);
const resultado = sumar4(4);
const resultado2 = sumar(1)(2)(3)(4);
console.log(resultado); // 20
console.log(resultado2); // 20
```

- en el ejemplo anterior, hay una serie de funciones anidadas, todas las cuales tienen acceso al alcance de las funciones externas.
- en este contexto, podemos decir que los cierres tienen acceso a *todos* los ámbitos de funciones externas.

Los cierres también pueden capturar variables en ámbitos de bloque y ámbitos de módulo.

- por ejemplo, lo siguiente crea un cierre sobre la **variable de ámbito de bloque** `y` :

```
function exterior() {
  const x = 5;
  if (Math.random() > 0.5) {
    const y = 6;
    return () => console.log(x, y);
  }
}

exterior()(); // Saca por consola: 5 6
```

Los cierres sobre **módulos** pueden ser más interesantes.

```
// miModulo.js
let x = 5;
export const getX = () => x;
export const setX = (valor) => {  x = valor; }
```

- aquí, el módulo exporta un par de **funciones getter-setter**, que se cierran sobre la **variable del ámbito del módulo** `x`
- incluso cuando no se puede acceder directamente a `x`desde otros módulos, se puede leer y escribir con las funciones.

```
import { getX, setX } from "./miModulo.js";

console.log(getX()); // 5
setX(6);
console.log(getX()); // 6
```

Los cierres también pueden cerrar sobre valores importados, que se consideran **enlaces activos** (***live bindings***), porque cuando cambia el valor original, el valor importado cambia en consecuencia.

```
// miModulo.js
export let x = 1;
export const setX = (valor) => { x = valor; }

// creadorDeCierre.js
import { x } from "./miModulo.js";
export const getX = () => x;   // Close over an imported live binding


import { getX } from "./creadorDeCierre.js";
import { setX } from "./miModulo.js";

console.log(getX()); // 1
setX(2);
console.log(getX()); // 2
```

Crear cierres en bucles: un error común

Antes de la introducción de la palabra clave **`let`**, ocurría un problema común con los **cierres** cuando los creaba dentro de un bucle.

- para demostrarlo, considera el siguiente código de ejemplo.

```
<p id="ayuda">Aquí aparecerán notas útiles</p>
<p>Correo electrónico: <input type="text" id="direccionEmail" name="email" /></p>
<p>Nombre: <input type="texto" id="nombre" nombre="nombre" /></p>
<p>Edad: <input type="text" id="edad" name="edad" /></p>

function mostrarAyuda(ayuda) {
  document.getElementById('ayuda').textContent = ayuda;
}

function configurarAyuda() {
  var textoAyuda = [
    { id: 'direccionEmail', ayuda: 'Tu dirección de email' },
    { id: 'nombre',         ayuda: 'Tu nombre completo' },
    { id: 'edad',           ayuda: 'Tu edad (debes ser mayor de 16 años)' },
  ];

  for (var i = 0; i < textoAyuda.length; i++) {
    // el culplable del problema es el uso de var en la declaración siguiente
    var elemento = textoAyuda[i];
    document.getElementById(elemento.id).onfocus = function () { // esta función es un cierre
      mostrarAyuda(elemento.ayuda);
    };
  }
}

configurarAyuda();
```

Si ejecutas el código anterior:

- el array **`textoAyuda`** define tres sugerencias útiles, cada una asociada con el **`id`** de un campo de entrada en el documento.
- el bucle recorre estas definiciones, **conectando** un evento **`onfocus`** a cada una que muestra el método de ayuda asociado.
- si pruebas este código, verás que no funciona como se esperaba: no importa en qué campo se centre, se mostrará el mensaje de ayuda sobre la edad.
  - la razón de esto es que las funciones asignadas a **`onfocus`** son cierres:
    - consisten en la definición de la función
    - y el entorno capturado del alcance de la función **`configurarAyuda`**.
  - el bucle crea 3 cierres:
    - pero cada uno comparte el mismo entorno léxico único, que tiene una variable con valores cambiantes (**`elemento`**).
    - la variable **`elemento`** se declara con **`var`** y, por lo tanto, tiene un alcance de función debido a la elevación (*hoisting*).
    - el valor de **`elemento.ayuda`** se determina cuando se ejecutan las devoluciones de llamada **`onfocus`**.
    - debido a que el bucle ya ha seguido su curso en ese momento, el objeto de la variable del elemento (compartido por los tres cierres) se ha dejado apuntando a la última entrada en la lista de texto de ayuda.

Una solución en este caso es usar más cierres: en particular, usar una **fábrica de funciones** como se describió anteriormente:

```
function mostrarAyuda(ayuda) {
  document.getElementById('ayuda').textContent = ayuda;
}

function contruirCallbackParaAyuda(ayuda) {
  return function () {
    mostrarAyuda(ayuda);
  };
}

function configurarAyuda() {
  var textoAyuda = [
    { id: 'direccionEmail', ayuda: 'Tu dirección de email' },
    { id: 'nombre',         ayuda: 'Tu nombre completo' },
    { id: 'edad',           ayuda: 'Tu edad (debes ser mayor de 16 años)' },
  ];

  for (var i = 0; i < textoAyuda.length; i++) {
    var elemento = textoAyuda[i];
```

```
    document.getElementById(elemento.id).onfocus = contruirCallbackParaAyuda(elemento.ayuda);
  }
}

configurarAyuda();
```

Si ejecutas el código anterior, funcionará como se esperaba:

- en lugar de que todas las devoluciones de llamada compartan un solo **entorno léxico**, la función `contruirCallbackParaAyuda` crea un nuevo entorno léxico para cada devolución de llamada, en el que la ayuda se refiere a la cadena correspondiente del array **`textoAyuda`**.

Otra forma de escribir lo anterior usando **cierres anónimos** es:

```
function mostrarAyuda(ayuda) {
  document.getElementById('ayuda').textContent = ayuda;
}

function configurarAyuda() {
  var textoAyuda = [
    { id: 'direccionEmail', ayuda: 'Tu dirección de email' },
    { id: 'nombre',         ayuda: 'Tu nombre completo' },
    { id: 'edad',           ayuda: 'Tu edad (debes ser mayor de 16 años)' },
  ];

  for (var i = 0; i < textoAyuda.length; i++) {
    (function () {
      var elemento = textoAyuda[i];
      document.getElementById(elemento.id).onfocus = function () {
        mostrarAyuda(elemento.ayuda);
      };
    })();    // Enlazamos inmediatamente el "eventListener" con el valor actual de elemento.ayuda
  }
}

configurarAyuda();
```

Ahora bien, si no quiere usar cierres, puede usar la palabra clave `let` o `const`:

```
function mostrarAyuda(ayuda) {
  document.getElementById('ayuda').textContent = ayuda;
}

function configurarAyuda() {
  var textoAyuda = [
    { id: 'direccionEmail', ayuda: 'Tu dirección de email' },
    { id: 'nombre',         ayuda: 'Tu nombre completo' },
    { id: 'edad',           ayuda: 'Tu edad (debes ser mayor de 16 años)' },
  ];

  for (let i = 0; i < textoAyuda.length; i++) {
    const elemento = textoAyuda[i];
    document.getElementById(elemento.id).onfocus = () => { mostrarAyuda(elemento.ayuda); };
  }
}

configurarAyuda();
```

- este ejemplo usa `const` en lugar de `var`, por lo que cada cierre vincula la **variable de ámbito de bloque**, lo que significa que no se requieren cierres adicionales.

Otra alternativa podría ser usar `forEach` () para iterar sobre el array **`textoAyuda`** y adjuntar un "listener" a cada **`<input>`**, como se muestra:

```
function mostrarAyuda(ayuda) {
  document.getElementById('ayuda').textContent = ayuda;
}
```

```
function configurarAyuda() {
  var textoAyuda = [
    { id: 'direccionEmail', ayuda: 'Tu dirección de email' },
    { id: 'nombre',         ayuda: 'Tu nombre completo' },
    { id: 'edad',           ayuda: 'Tu edad (debes ser mayor de 16 años)' },
  ];

  textoAyuda.forEach(function (texto) {
    document.getElementById(texto.id).onfocus = function (){ mostrarAyuda(texto.ayuda); };
  });
}

configurarAyuda();
```

Consideraciones de rendimiento

Como se mencionó anteriormente, cada **instancia de función** administra su propio **alcance** y **cierre**.

- por lo tanto, no es aconsejable crear funciones innecesariamente dentro de otras funciones si no se necesitan cierres para una tarea en particular, ya que afectará negativamente **el rendimiento del script** tanto en términos de **velocidad de procesamiento** como de **consumo de memoria**.
- por ejemplo, al crear un nuevo objeto/clase, los métodos normalmente deben asociarse al **prototipo del objeto** en lugar de definirse en el **constructor del objeto**.
- la razón es que cada vez que se llama al constructor, los métodos se reasignan (es decir, para cada creación de objeto).

Considere el siguiente caso:

```
function MiObjeto(nombre, mensaje) {
  this.nombre = nombre.toString();
  this.mensaje = mensaje.toString();

  this.getNombre  = function () { return this.nombre;  };
  this.getMensaje = function () { return this.mensaje; };
}
```

Debido a que el código anterior no aprovecha los beneficios de usar cierres en este caso particular, podríamos reescribirlo para evitar el uso de cierres de la siguiente manera:

A) Redefinir el prototipo (no recomendado):

```
function MiObjeto(nombre, mensaje) {
  this.nombre = nombre.toString();
  this.mensaje = mensaje.toString();
}

MiObjeto.prototype = {
  getNombre()  { return this.nombre;  },
  getMensaje() { return this.mensaje; },
};
```

Sin embargo, no se recomienda redefinir el prototipo.

B) Anexando al prototipo existente:

En cambio, el siguiente ejemplo se agrega al prototipo existente:

```
function MiObjeto(nombre, mensaje) {
  this.nombre = nombre.toString();
  this.mensaje = mensaje.toString();
}

MiObjeto.prototype.getNombre  = function () { return this.nombre; };
MiObjeto.prototype.getMensaje = function () { return this.mensaje; };
```

En los dos ejemplos anteriores, el **prototipo heredado** puede ser compartido por todos los objetos y no es necesario que las definiciones de métodos ocurran en cada creación de objetos.