Esmeraldas 07 de Julio del 2024

La desestructuración

La sintaxis de **desestructuración** es una expresión de JavaScript que permite desempacar valores de arreglos o propiedades de objetos en distintas variables.

Pruébalo

La fuente de este ejemplo interactivo se almacena en un repositorio de GitHub. Si deseas contribuir al proyecto de ejemplos interactivos,
clona https://github.com/mdn/interactive-examples y envíanos una solicitud de extracción.

Sintaxis

JSCopy to Clipboard

```
let a, b, rest;
[a, b] = [10, 20];
console.log(a); // 10
console.log(b); // 20

[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]

({ a, b } = { a: 10, b: 20 });
console.log(a); // 10
console.log(b); // 20

// Propuesta de etapa 4 (terminada)
({ a, b, ...rest } = { a: 10, b: 20, c: 30, d: 40 });
console.log(a); // 10
console.log(b); // 20
console.log(rest); // {c: 30, d: 40}
```

Descripción

Las expresiones de objetos y arreglos literales proporcionan una manera fácil de crear paquetes de datos *ad hoc*.

JSCopy to Clipboard

```
const x = [1, 2, 3, 4, 5];
```

La desestructuración utiliza una sintaxis similar, pero en el lado izquierdo de la asignación para definir qué valores desempacar de la variable origen.

JSCopy to Clipboard

```
const x = [1, 2, 3, 4, 5];
```

```
const [y, z] = x;
console.log(y); // 1
console.log(z); // 2
```

Esta capacidad es similar a las características presentes en lenguajes como Perl y Python.

Ejemplos

Desestructuración de arreglos

Asignación básica de variables

JSCopy to Clipboard

```
const foo = ["one", "two", "three"];

const [red, yellow, green] = foo;
console.log(red); // "one"
console.log(yellow); // "two"
console.log(green); // "three"
```

Asignación separada de la declaración

A una variable se le puede asignar su valor mediante una desestructuración separada de la declaración de la variable.

JSCopy to Clipboard

```
let a, b;

[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

Valores predeterminados

A una variable se le puede asignar un valor predeterminado, en el caso de que el valor desempacado del arreglo sea undefined.

JSCopy to Clipboard

```
let a, b;

[a = 5, b = 7] = [1];
console.log(a); // 1
console.log(b); // 7
```

Intercambio de variables

Los valores de dos variables se pueden intercambiar en una expresión de desestructuración.

Sin desestructurar la asignación, intercambiar dos valores requiere una variable temporal (o, en algunos lenguajes de bajo nivel, el algoritmo del truco XOR-swap).

JSCopy to Clipboard

```
let a = 1;
let b = 3;

[a, b] = [b, a];
console.log(a); // 3
console.log(b); // 1

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

Analizar un arreglo devuelto por una función

Siempre ha sido posible devolver un arreglo desde una función. La desestructuración puede hacer que trabajar con un valor de retorno de arreglo sea más conciso.

En este ejemplo, `f()` devuelve los valores `[1, 2]` como su salida, que se puede procesar en una sola línea con desestructuración.

JSCopy to Clipboard

```
function f() {
  return [1, 2];
}

let a, b;
[a, b] = f();
console.log(a); // 1
console.log(b); // 2
```

Ignorar algunos valores devueltos

Puedes ignorar los valores de retorno que no te interesan:

JSCopy to Clipboard

```
function f() {
  return [1, 2, 3];
}

const [a, , b] = f();
console.log(a); // 1
console.log(b); // 3

const [c] = f();
console.log(c); // 1
```

También puedes ignorar todos los valores devueltos:

JSCopy to Clipboard

```
[, ,] = f();
```

Asignar el resto de un arreglo a una variable

Al desestructurar un arreglo, puedes desempacar y asignar la parte restante a una variable usando el patrón rest o:

JSCopy to Clipboard
```
const [a, ...b] = [1, 2, 3];
console.log(a); // 1
console.log(b); // [2, 3]
```

Ten en cuenta que se lanzará un SyntaxError si se usa una coma final en el lado derecho con un elemento rest o:

JSCopy to Clipboard
```
const [a, ...b,] = [1, 2, 3];

// SyntaxError: el elemento rest no puede tener una coma al final
// Siempre considera usar el operador rest como último elemento
```

Desempacar valores coincidentes con una expresión regular

Cuando el método de expresión regular exec() encuentra una coincidencia, devuelve un arreglo que contiene primero toda la parte coincidente de la cadena y luego las partes de la cadena que coinciden con cada grupo entre paréntesis en la expresión regular. La desestructuración te permite desempacar fácilmente las partes de este arreglo, ignorando la coincidencia completa si no es necesaria.

JSCopy to Clipboard
```
function parseProtocol(url) {
  const parsedURL = /^(\w+)\:\/\/([^\/]+)\/(.*)$/.exec(url);
  if (!parsedURL) {
    return false;
  }
  console.log(parsedURL);
  // ["https://developer.mozilla.org/es/Web/JavaScript",
     "https", "developer.mozilla.org", "es/Web/JavaScript"]

  const [, protocol, fullhost, fullpath] = parsedURL;
  return protocol;
}

console.log(parseProtocol('https://developer.mozilla.org/es/Web/JavaScript'));
// "https"
```

Desestructuración de objetos

Asignación básica

JSCopy to Clipboard
```
const user = {
  id: 42,
  is_verified: true,
};

const { id, is_verified } = user;
```

```
console.log(id); // 42
console.log(is_verified); // true
```

Asignación sin declaración

A una variable se le puede asignar su valor con desestructuración separada de su declaración.

JSCopy to Clipboard
```
let a, b;

({ a, b } = { a: 1, b: 2 });
```
Nota: Los paréntesis (...) alrededor de la declaración de asignación son obligatorios cuando se usa la desestructuración de un objeto literal sin una declaración.
{a, b} = {a: 1, b: 2} no es una sintaxis independiente válida, debido a que {a, b} en el lado izquierdo se considera un bloque y no un objeto literal.
Sin embargo, ({a, b} = {a: 1, b: 2}) es válido, al igual que const {a, b} = {a: 1, b: 2}
tu expresión (…) debe estar precedida por un punto y coma o se puede usar para ejecutar una función en la línea anterior.

Asignar a nuevos nombres de variable

Una propiedad se puede desempacar de un objeto y asignar a una variable con un nombre diferente al de la propiedad del objeto.

JSCopy to Clipboard
```
const o = { p: 42, q: true };
const { p: foo, q: bar } = o;

console.log(foo); // 42
console.log(bar); // true
```

Aquí, por ejemplo, const {p: foo} = o toma del objeto o la propiedad llamada p y la asigna a una variable local llamada foo.

Valores predeterminados

A una variable se le puede asignar un valor predeterminado, en el caso de que el valor desempacado del objeto sea undefined.

JSCopy to Clipboard
```
const { a = 10, b = 5 } = { a: 3 };

console.log(a); // 3
console.log(b); // 5
```

Asignar nombres a nuevas variables y proporcionar valores predeterminados

Una propiedad puede ser ambas

- Desempacada de un objeto y asignada a una variable con un nombre diferente.

- Se le asigna un valor predeterminado en caso de que el valor desempacado sea undefined.

JSCopy to Clipboard
```
const { a: aa = 10, b: bb = 5 } = { a: 3 };

console.log(aa); // 3
console.log(bb); // 5
```

Desempacar campos de objetos pasados como parámetro de función

JSCopy to Clipboard
```
const user = {
  id: 42,
  displayName: "jdoe",
  fullName: {
    firstName: "John",
    lastName: "Doe",
  },
};

function userId({ id }) {
  return id;
}

function whois({ displayName, fullName: { firstName: name } }) {
  return `${displayName} es ${name}`;
}

console.log(userId(user)); // 42
console.log(whois(user)); // "jdoe es John"
```

Esto desempaca el id, displayName y firstName del objeto user y los imprime.

Establecer el valor predeterminado de un parámetro de función

JSCopy to Clipboard
```
function drawChart({
  size = "big",
  coords = { x: 0, y: 0 },
  radius = 25,
} = {}) {
  console.log(size, coords, radius);
  // haz un dibujo de gráfico
}

drawChart({
  coords: { x: 18, y: 30 },
  radius: 30,
});
```

Nota: En la firma de la función para **drawChart** anterior, el lado izquierdo desestructurado se asigna a un objeto literal vacío en el lado derecho: {size = 'big', coords = {x: 0, y: 0}, radius = 25} = {}. También podrías haber escrito la función sin la asignación del lado derecho. Sin embargo, si omites la asignación del lado derecho, la función buscará al menos un argumento para ser proporcionado cuando se invoca, mientras que en su forma actual, simplemente puedes llamar a **drawChart()** sin proporcionar ningún

parámetro. El diseño actual es útil si deseas poder llamar a la función sin proporcionar ningún parámetro, el otro puede ser útil cuando deseas asegurarte de que se pase un objeto a la función.

Desestructuración de arreglos y objetos anidados

JSCopy to Clipboard

```
const metadata = {
 title: "Scratchpad",
 translations: [
  {
   locale: "de",
   localization_tags: [],
   last_edit: "2020-08-29T08:43:37",
   url: "/de/docs/Tools/Scratchpad",
   title: "JavaScript-Umgebung",
  },
 ],
 url: "/es/docs/Tools/Scratchpad",
};

let {
 title: englishTitle, // renombrar
 translations: [
  {
   title: localeTitle, // renombrar
  },
 ],
} = metadata;

console.log(englishTitle); // "Scratchpad"
console.log(localeTitle); // "JavaScript-Umgebung"
```

Iteración "for...of" y desestructuración

JSCopy to Clipboard

```
const people = [
 {
  name: "Mike Smith",
  family: {
   mother: "Jane Smith",
   father: "Harry Smith",
   sister: "Samantha Smith",
  },
  age: 35,
 },
 {
  name: "Tom Jones",
  family: {
   mother: "Norah Jones",
   father: "Richard Jones",
   brother: "Howard Jones",
  },
  age: 25,
 },
];
```

```
for (const {
  name: n,
  family: { father: f },
} of people) {
  console.log("Nombre: " + n + ", Padre: " + f);
}

// "Nombre: Mike Smith, Padre: Harry Smith"
// "Nombre: Tom Jones, Padre: Richard Jones"
```

Nombres de propiedades de objetos calculados y desestructuración

Los nombres de propiedad calculados, como en un Objeto literal, se pueden usar con la desestructuración.

JSCopy to Clipboard

```
let key = "z";
let { [key]: foo } = { z: "bar" };

console.log(foo); // "bar"
```

Rest en la desestructuración de objetos

La propuesta Propiedades rest/propagación para ECMAScript (etapa 4) agrega la sintaxis rest para desestructurar. Las propiedades de rest recopilan las claves de propiedades enumerables restantes que aún no han sido seleccionadas por el patrón de desestructuración.

JSCopy to Clipboard

```
let { a, b, ...rest } = { a: 10, b: 20, c: 30, d: 40 };
a; // 10
b; // 20
rest; // { c: 30, d: 40 }
```

Identificador de JavaScript no válido como nombre de propiedad

La desestructuración se puede utilizar con nombres de propiedad que no son identificadores válidos en JavaScript proporcionando un identificador alternativo que sea válido.

JSCopy to Clipboard

```
const foo = { "fizz-buzz": true };
const { "fizz-buzz": fizzBuzz } = foo;

console.log(fizzBuzz); // "true"
```

Desestructuración combinada de arreglos y objetos

La desestructuración de arreglos y objetos se puede combinar. Supongamos que deseas manipular el tercer elemento del siguiente arreglo props, y luego deseas la propiedad name en el objeto, puedes hacer lo siguiente:

JSCopy to Clipboard

```
const props = [
  { id: 1, name: "Fizz" },
  { id: 2, name: "Buzz" },
  { id: 3, name: "FizzBuzz" },
];

const [, , { name }] = props;

console.log(name); // "FizzBuzz"
```

Se busca la cadena de prototipos al desestructurar el objeto

Al deconstruir un objeto, si no se accede a una propiedad en sí misma, continuará buscando a lo largo de la cadena de prototipos.

JSCopy to Clipboard

```
let obj = { self: "123" };
obj.__proto__.prot = "456";
const { self, prot } = obj;
// self "123"
// prot "456" (Acceso a la cadena de prototipos)
```

JavaScript rest vs. operador de propagación (spread): ¿Cuál es la diferencia?

[Traductor: Keiler Guardo Herrera]

Autor: Oluwatobi Sofela (Inglés)

Original article: [JavaScript Rest vs Spread Operator – What's the Difference?]

JavaScript utiliza tres puntos (. . .) tanto para los operadores de rest como para propagación. Pero estos dos operadores no son lo mismo. La principal diferencia entre rest y el operador propagación es que el operador rest pone el resto de algunos valores específicos suministrados por el usuario en un arreglo de JavaScript. Pero la progración expande los iterables en elementos individuales.

Por ejemplo, considera este código que utiliza rest para encerrar algunos valores en un arreglo:

```
// Utiliza rest para encerrar el resto de los valores específicos suministrados por el usuario en un arreglo:
function miBiografia(primerNombre, apellido, ...otraInformacion) {
  return otraInformacion;
}

// Llama la función miBiografia pasando cinco argumentos a sus parámetros:
miBiografia("Oluwatobi", "Sofela", "CodeSweetly", "Desarrollador Web", "Masculino");

// La llamada de arriba devolverá:
["CodeSweetly", "Desarrollador Web", "Masculino"]
```

Pruébalo en StackBlitz

En el ejemplo anterior, utilizamos el parámetro rest en `...otraInformacion` para poner `"CodeSweetly"`, `"Desarrollador Web"` y `"Masculino"` en un un arreglo.

Ahora, considera este ejemplo del operador de propagación:

```
// Define una función con tres parámetros:
function miBiografia(primerNombre, apellido, compania) {
  return `${primerNombre} ${apellido} dirige ${compania}`;
}

// Utiliza propagación para expandir los elementos de un arreglo de elementos individuales:
miBiografia(...["Oluwatobi", "Sofela", "CodeSweetly"]);

// La llamada de arriba devolverá:
“Oluwatobi Sofela dirige CodeSweetly”
```

Pruébalo en StackBlitz
En el fragmento anterior, hemos utilizado el operador de propagación (. . .) para propagar el contenido de `["Oluwatobi", "Sofela", "CodeSweetly"]` entre los parámetros de `miBiografia()`. No te preocupes si aún no entiendes el operador rest o el operador de propagación. ¡Este artículo lo tiene cubierto!

En las siguientes secciones, discutiremos cómo trabaja el operador rest y operador de propagación en JavaScript.

Así que, sin más preámbulos, empecemos con el operador rest.

¿Qué es exactamente el operador rest?

El **operador rest** se utiliza para poner el resto de algunos valores específicos suministrados por el usuario en un arreglo de JavaScript. Así, por ejemplo, aquí está la sintaxis de rest:

```
...tusValores
```

Los tres puntos (. . .) en el fragmento anterior simbolizan el operador rest.
El texto que aparece después del operador rest hace referencia a los valores que deseas incluir en un arreglo. Sólo se puede utilizar antes del último parámetro en la definición de una función.

Para entender mejor la sintaxis, veamos cómo funciona rest con las funciones de JavaScript.

¿Cómo funciona el operador rest en una función?

En las funciones de JavaScript, rest se utiliza como antepuesto del último parámetro de la función.

He aquí un ejemplo:

```
// Define una función con dos parámetros regulares y un parámetro rest:
function miBiografia(primerNombre, apellido, ...otraInformacion) {
  return otraInformacion;
}
```

El operador rest (. . .) le indica al ordenador que añada cualquier argumento en `otraInformacion` suministrado por el usuario en un arreglo. A su vez, asigna ese arreglo al parámetro `otraInformacion`.

Como tal, llamamos a `...otraInformacion` un parámetro rest.
Nota: Los argumentos son valores opcionales que se pueden pasar al parámetro de una función a través de un invocador.

JSON para principiantes: qué es, para qué sirve y ejemplos

11 DE MAYO DE 2023
INIGO

JSON es un formato de intercambio de datos que se utiliza para transmitir información entre diferentes sistemas. Es una forma sencilla y legible de estructurar datos para su uso en aplicaciones web y móviles. Algunos ejemplos de uso incluyen el intercambio de información de productos en línea, datos de usuarios y mensajes entre aplicaciones.

Descubre cómo el formato JSON puede mejorar la eficacia de tus estrategias de marketing

¿Te has preguntado alguna vez cómo los datos se transmiten a través de la web? Bueno, la mayoría de las veces, se utilizan formatos de intercambio de datos para que los datos puedan ser compartidos fácilmente entre diferentes aplicaciones. Uno de esos formatos es JSON, y hoy nosotros te explicaremos de qué se trata y cómo puede mejorar la eficacia de tus estrategias de marketing.

JSON significa «JavaScript Object Notation» y es un formato de intercambio de datos ligero y fácil de leer que se utiliza para transmitir datos estructurados. JSON es muy popular en la web, ya que es fácilmente interpretable por los navegadores web y otros programas.

1. Descubre cómo el formato JSON puede mejorar la eficacia de tus estrategias de marketing: ¿Cómo funciona?

JSON utiliza una sintaxis de objetos que se parece mucho a la sintaxis de los objetos en JavaScript. Los objetos en JSON se escriben como pares clave-valor y se separan por comas. Por ejemplo, aquí hay un objeto JSON que representa un producto:

{
«nombre»: «iPhone 12»,
«precio»: 999,
«disponible»: true
}

En este ejemplo, «nombre», «precio» y «disponible» son claves, mientras que «iPhone 12», 999 y true son los valores correspondientes. Los objetos JSON pueden anidarse dentro de otros objetos, lo que permite estructurar datos complejos.

2. Descubre cómo el formato JSON puede mejorar la eficacia de tus estrategias de marketing: ¿Para qué sirve?

JSON se utiliza para intercambiar datos entre diferentes aplicaciones web. Los datos pueden ser enviados desde un servidor a un navegador web o viceversa, y pueden ser utilizados para actualizar el contenido de una página web en tiempo real. También se utiliza en APIs (interfaces de programación de aplicaciones) para que los desarrolladores puedan acceder y utilizar los datos de una aplicación.

En términos de marketing, JSON puede ser utilizado para recopilar datos de los usuarios, como sus preferencias de compra o su historial de navegación, y utilizar estos datos para personalizar la experiencia del usuario en tu sitio web. También puede ser utilizado para recopilar datos de redes sociales y otros sitios web para realizar análisis de marketing y segmentación de audiencia.

3. Descubre cómo el formato JSON puede mejorar la eficacia de tus estrategias de marketing: Ejemplos

Aquí hay algunos ejemplos de cómo JSON puede ser utilizado en el marketing:

– Personalización del contenido: Utiliza JSON para recopilar información sobre las preferencias de tus usuarios y personalizar el contenido de tu sitio web de acuerdo con sus intereses.
– Análisis de marketing: Utiliza JSON para recopilar datos de las redes sociales y otros sitios web y realizar análisis de marketing para entender mejor a tu audiencia y segmentarla de manera más efectiva.
– Integración de aplicaciones: Utiliza JSON para integrar diferentes aplicaciones de marketing, como plataformas de publicidad, análisis web y herramientas de automatización de correo electrónico, para mejorar la eficacia de tus estrategias de marketing.

En resumen, JSON es un formato de intercambio de datos popular y fácil de leer que se utiliza para transmitir datos estructurados en la web. Puede mejorar la eficacia de tus estrategias de marketing al permitir la personalización del contenido, el análisis de marketing y la integración de aplicaciones. ¡Así que no dudes en incorporar JSON a tu arsenal de herramientas de marketing!

En conclusión, JSON es un formato de intercambio de datos ligero y fácil de leer que se utiliza para transmitir información entre aplicaciones. Se utiliza en una amplia variedad de aplicaciones, desde el intercambio de datos en la web hasta la transferencia de información entre dispositivos móviles y de escritorio. **La importancia de JSON radica en su facilidad de uso y su capacidad para transmitir datos de manera eficiente.** Algunos ejemplos de su uso incluyen la visualización de datos en páginas web dinámicas, la transferencia de información en aplicaciones móviles y la integración de sistemas de información empresarial.

¿Sabes que es una funcion en javascript y cuáles son las funciones anónimas?

JavaScript es un lenguaje de programación con una sintaxis basada en objetos. Uno de estos objetos son las funciones en javascript , fundamentales para ejecutar cualquier acción que queramos en nuestro código. En este post, te enseñaremos **algunas particularidades de las funciones**

anónimas en JavaScript, aquellas a las que no le declaramos nombre.

¿Qué encontrarás en este post?

Un poco de contexto

En nuestro post sobre los métodos *querySelector* y *querySelectorAll*, hemos usado las siguientes líneas de código para explicar el impacto de este segundo método:

```
const buttonListElement = document.querySelectorAll

for (const button of buttonListElement) {

button.addEventListener (‘click’, ( ) => {

drawTime ( );

});

}

function drawTime ( ) {

document.getElementById («demo»).innerHTML = new Date ( );

}
```

Como puedes notar, en las anteriores líneas hemos decidido manejar eventos imperativos en *frontend* JavaScript e **insertar una *arrow function* anónima que devuelve la función *drawTime.*** A continuación, te mostramos la otra manera de escribir este código.

```
for (const button of buttonListElement) {

button.addEventListener (‘click’, drawTime);

}
```

En la sección de código anterior estamos insertando directamente la función *drawTime* como parámetro del método *addEventListener.* Esto quiere decir que el manejador del evento *click* será *drawTime.* La tercera opción para escribir este código, similar a la primera, es **crear una función anónima con la palabra clave *function***:

```
for (const button of buttonListElement) {

button.addEventListener (‘click’, function ( ) {

drawTime ( );

});

}
```

Aunque estas tres formas de escribir un evento funcionan en nuestro contexto, debemos tener mucho cuidado con su uso. ¿Por qué? Pues porque **las tres funciones se relacionan de maneras distintas con la palabra clave *this.***

Funciones anónimas en JavaScript y *this*

Las funciones anónimas en JavaScript son aquellas que no han sido declaradas con un nombre. En este lenguaje de programación, podemos declarar este tipo de elemento usando cualquiera de los modos de escribir funciones.

Es decir, podemos declarar funciones anónimas en JavaScript con *arrow function*, un modo de escritura que siempre es anónimo:

```
( ) => { }
```

O con la palabra clave *function*:

function ( ) { }

Como mencionamos antes, estas dos maneras de escribir funciones anónimas en JavaScript implican dos aproximaciones diferentes al entendimiento de *this.*

En la función de javascript anónima de tipo *arrow function,* la palabra clave *this* toma el valor del *scope* superior. Por su parte, en la función anónima de tipo *function*, el valor pertenece a quien ejecuta esta función. Por esto **es muy importante reconocer las distintas maneras de escribir funciones anónimas en JavaScript**, pues tu código puede tener resultados muy diferentes según como determines la función.

Ten en cuenta que en nuestro ejemplo anterior no hemos visto una diferencia en el resultado porque nuestra función *drawTime* no usa la palabra clave *this* y estamos atacando directamente al DOM.

Funciones no anónimas

También cabe resaltar que **el efecto de escribir una función cambia incluso en las funciones declaradas por nombre.** Supongamos que tenemos las siguientes dos maneras de ejecutar la función *drawTime* que hemos declarado antes, pero con algún valor en el parámetro:

button.addEventListener (‘click’, drawTime)

drawTime ( );

Las dos formas anteriores de ejecutar la función son muy distintas, porque en el primer caso la función *drawTime* **se asigna como consecuencia o respuesta a un evento que sucede.** Por ello, se ejecutará desde un contexto distinto al ejecutado en la segunda línea de código. En este sentido, el valor entre paréntesis de la función *drawTime* sería el *evento*:

function drawTime (event) {

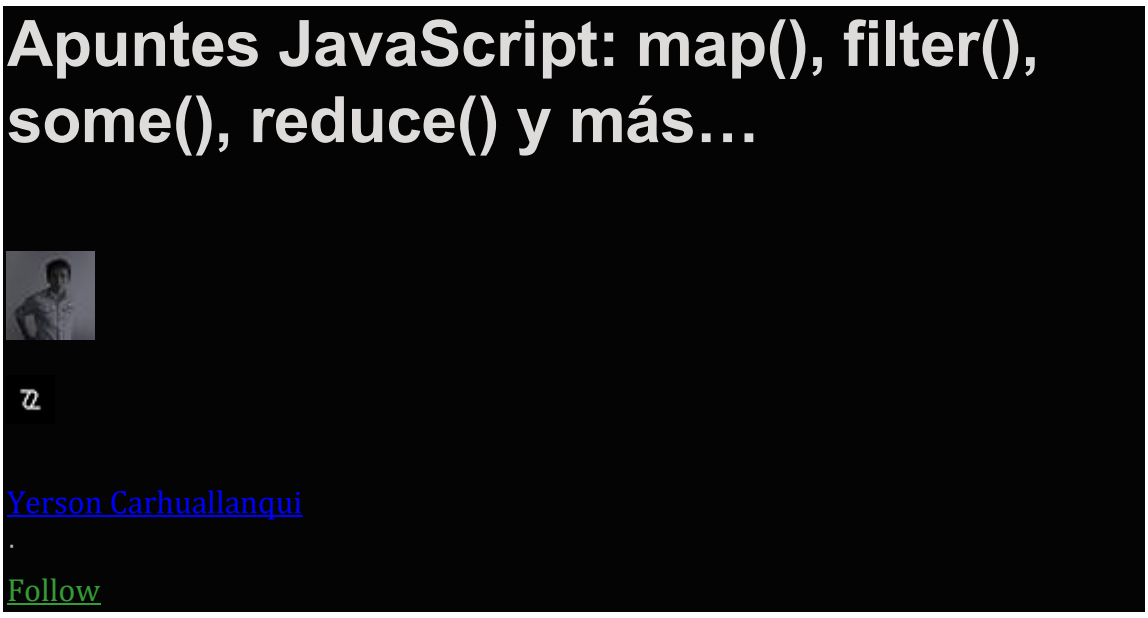
```
document.getElementById (‘demo’).innerHTML = new Date

}
```

Ten presente que, aunque en las líneas anteriores hemos nombrado al parámetro automático como *event,* **puedes llamarlo como quieras y representará el evento recibido igualmente.** Sea cual sea el nombre que le pongas, este objeto nos dará información sobre el evento recibido en la función, como lo son el *path* y el *target.* Si quieres conocer más sobre este objeto, puedes insertar el comando *console.log (event)* dentro de la función *drawTime*. También puedes leer nuestro post sobre *event.target* en JavaScript.

¿Qué sigue?

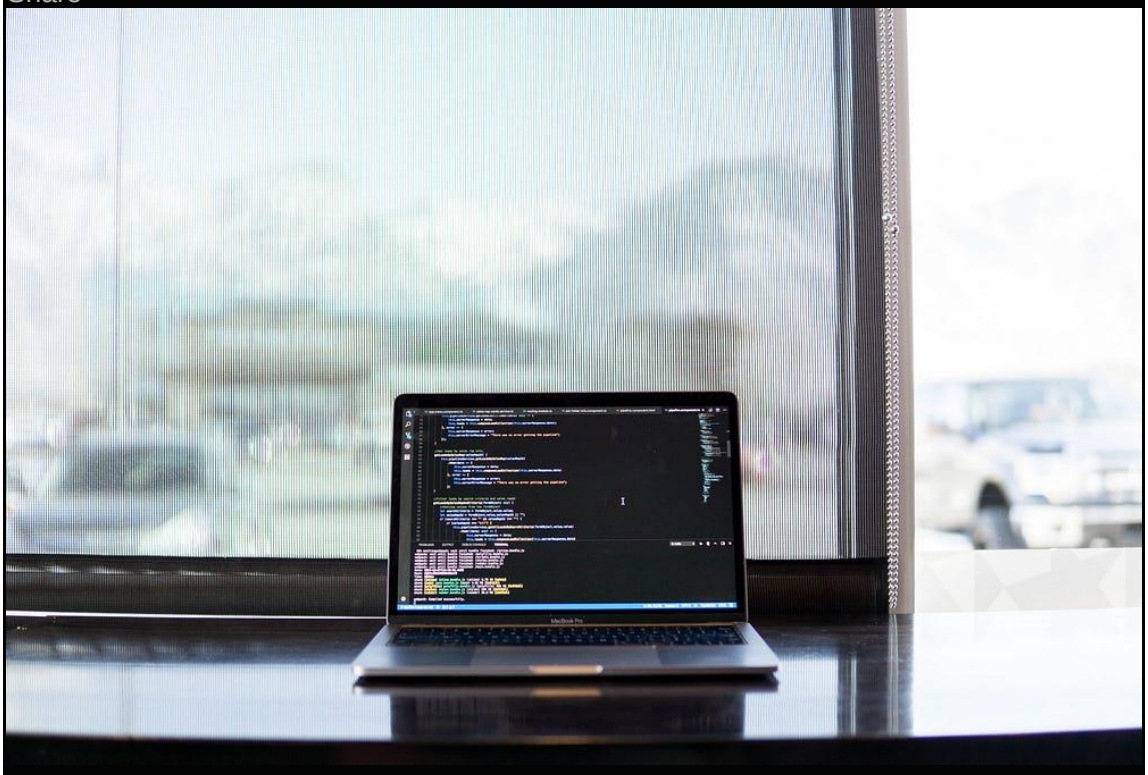
Ahora que sabes qué son y cómo funcionan las funciones anónimas en JavaScript, te invitamos a **seguir aprendiendo sobre las posibilidades de este y otros lenguajes de programación** en nuestro Desarrollo Web Full Stack Bootcamp. En este espacio de formación intensiva aprenderás a dominar diversas herramientas y lenguajes como HTML, CSS, JavaScript y JSX. **¡No te lo pierdas y pide información!**



Published in

Zurvin

2 min read
.
Oct 19, 2019
Share



Permítame compartirles algunos apuntes que probablemente ya lo han estado usando. Son increíblemente útiles y hacen nuestro código mas ‘limpio’.

- **No olvidar:** Ninguno altera el array inicial.

map()

Ojo al piojo: Si peques=[], map retorna []

```
/**
 * @return Array
 */
let peques = [🐣, 🐛, 👶];
let grandes = peques.map(crecer);
// => [🐓, 🦋, 👨]
```

.map()

filter()

Si peques=[], filter retorna []

```
/**
 * @return Array
 */
let peques = [🐣, 🐛, 👶];
let humanos = peques.filter(esHumano);
// => [👶]
```

.filter()

find()

Retorna el primer elemento encontrado. Si no encuentra coincidencias, retorna **undefined.**

```
/**
 * @return Value
 */
let peques = [🐣, 🐛, 👶];
let unHumano = peques.find(esHumano);
// => 👶
```

.find()

some()

Sólo retorna true/false, muy útil.

```
/**
 * @return Boolean
 */
let peques = [🐣, 🦠, 👶];
let existeHumano = peques.some(buscaHumano);
// => true
```

.some()

every()

Todo o nadie.

```
/**
 * @return Boolean
 */
let peques = [🐣, 🦠, 👶];
let todosTienenVida = peques.every(viven);
// => true
```

.every()

reduce()

Si peques=[] lanza **error de array vacío**.

.reduce()

```
// Caso práctico: Suma elementos de un array
arr.reduce((a, b) => a + b);
```

Si te sirve genial, y si ya lo sabías doblemente genial. Y si compartes, genial x3.