29º marzo 2021

Ámbito de Variable y el concepto "closure"

JavaScript es un lenguaje muy orientado a funciones. Nos da mucha libertad. Una función se puede crear en cualquier momento, pasar como argumento a otra función y luego llamar desde un lugar de código totalmente diferente más tarde.

Ya sabemos que una función puede acceder a variables fuera de ella.

Pero, ¿qué sucede si estas variables "externas" cambian desde que se crea una función? ¿La función verá los valores nuevos o los antiguos?

Y si una función se pasa como parámetro y se llama desde otro lugar del código, ¿tendrá acceso a las variables externas en el nuevo lugar?

Ampliemos nuestro conocimiento para comprender estos escenarios y otros más complejos.

Hablaremos de las variables let / const aquí

En JavaScript, hay 3 formas de declarar una variable: `let`, `const` (las modernas) y `var` (más antigua).

- En este artículo usaremos las variables `let` en los ejemplos.
- Las variables, declaradas con `const`, se comportan igual, por lo que este artículo también trata sobre `const`.
- El antiguo `var` tiene algunas diferencias notables, se tratarán en el artículo La vieja "var".

Bloques de código

Si una variable se declara dentro de un bloque de código `{...}`, solo es visible dentro de ese bloque.

Por ejemplo:

```
{
  // hacer un trabajo con variables locales que no deberían verse fuera
  let message = "Hello"; // solo visible en este bloque
  alert(message); // Hello
}

alert(message); // Error: el mensaje no se ha definido (undefined)
```

Podemos usar esto para aislar un fragmento de código que realiza su propia tarea, con variables que solo le pertenecen a él:

```
{
  // ver mensaje
  let message = "Hello";
  alert(message);
}

{
  // ver otro mensaje
  let message = "Goodbye";
  alert(message);
}
```

Habría un error sin bloques

Tenga en cuenta que sin bloques separados, habría un error, si usamos 'let' con el nombre de la variable existente:

```
// ver mensaje
let message = "Hello";
alert(message);

// ver otro mensaje
let message = "Goodbye"; // Error: la variable ya ha sido declarada
alert(message);
```

Para `if`, `for`, `while` y así sucesivamente, las variables declaradas en `{...}` de igual manera solo son visibles en el interior:

```
if (true) {
  let phrase = "Hello!";

  alert(phrase); // Hello!
}

alert(phrase); // ¡Error, no hay tal variable!
```

Aquí, después de que `if` termine, la `alerta` a continuación no verá la `phrase`, de ahí el error.

Eso es genial, ya que nos permite crear variables locales de bloque, específicas de una rama `if`.

De la misma manera que para los bucles `for` y `while`:

```
for (let i = 0; i < 3; i++) {
  // la variable i solo es visible dentro de este for
  alert(i); // 0, then 1, then 2
}

alert(i); // ¡Error, no hay tal variable!
```

Visualmente, `let i` está fuera de `{...}`. Pero la construcción `for` es especial aquí: la variable, declarada dentro de ella, se considera parte del bloque.

Funciones anidadas

Una función se llama "anidada" cuando se crea dentro de otra función.

Es fácilmente posible hacer esto con JavaScript.

Podemos usarlo para organizar nuestro código, así:

```
function sayHiBye(firstName, lastName) {

  // función anidada auxiliar para usar a continuación
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```

Aquí la función *anidada* `getFullName()` se hace por conveniencia. Puede acceder a las variables externas y, por lo tanto, puede devolver el nombre completo. Las funciones anidadas son bastante comunes en JavaScript.

Lo que es mucho más interesante, es que puede devolverse una función anidada: ya sea como propiedad de un nuevo objeto o como resultado en sí mismo. Luego se puede usar en otro lugar. No importa dónde, todavía tiene acceso a las mismas variables externas.

A continuación, `makeCounter` crea la función "contador "que devuelve el siguiente número en cada invocación:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

A pesar de ser simples, las variantes ligeramente modificadas de ese código tienen usos prácticos, por ejemplo, como random number generator para generar valores aleatorios para pruebas automatizadas.

¿Cómo funciona esto? Si creamos múltiples contadores, ¿serán independientes? ¿Qué está pasando con las variables aquí?

Entender tales cosas es excelente para el conocimiento general de JavaScript y beneficioso para escenarios más complejos. Así que vamos a profundizar un poco.

Ámbito o alcance léxico

> **¡Aquí hay dragones!**
>
> La explicación técnica en profundidad está por venir.
>
> Me gustaría evitar los detalles de lenguaje de bajo nivel, pero cualquier comprensión sin ellos sería insuficiente e incompleta, así que prepárate.

Para mayor claridad, la explicación se divide en múltiples pasos.

Paso 1. Variables

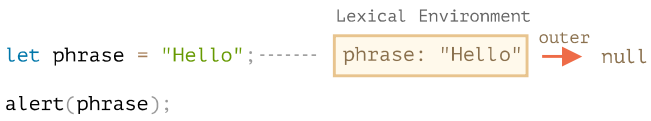
En JavaScript, todas las funciones en ejecución, el bloque de código `{...}` y el script en su conjunto tienen un objeto interno (oculto) asociado, conocido como *Alcance léxico*.

El objeto del alcance léxico consta de dos partes:

1. *Registro de entorno*: un objeto que almacena todas las variables locales como sus propiedades (y alguna otra información como el valor de `this`).
2. Una referencia al *entorno léxico externo*, asociado con el código externo.

Una "variable" es solo una propiedad del objeto interno especial, `Registro de entorno`. "Obtener o cambiar una variable" significa "obtener o cambiar una propiedad de ese objeto".

En este código simple y sin funciones, solo hay un entorno léxico:

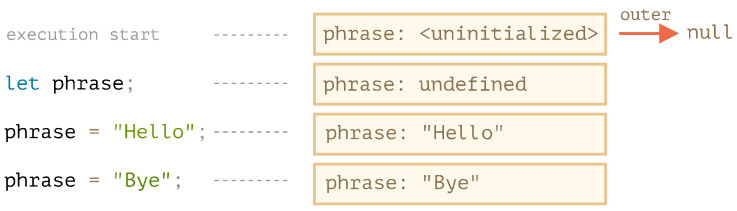


Este es el denominado entorno léxico *global*, asociado con todo el script.

En la imagen de arriba, el rectángulo significa Registro de entornos (almacén de variables) y la flecha significa la referencia externa. El entorno léxico global no tiene referencia externa, por eso la flecha apunta a `nulo`.

A medida que el código comienza a ejecutarse y continúa, el entorno léxico cambia.

Aquí hay un código un poco más largo:



Los rectángulos en el lado derecho demuestran cómo cambia el entorno léxico global durante la ejecución:

1. Cuando se inicia el script, el entorno léxico se rellena previamente con todas las variables declaradas. – Inicialmente, están en el estado "No inicializado". Ese es un estado interno especial, significa que el motor conoce la variable, pero no se puede hacer referencia a ella hasta que se haya declarado con `let`. Es casi lo mismo que si la variable no existiera.
2. Luego aparece la definición `let phrase`. Todavía no hay una asignación, por lo que su valor es `undefined`. Podemos usar la variable desde este punto en adelante.
3. `phrase` se le asigna un valor.
4. `phrase` cambia el valor.

Todo parece simple por ahora, ¿verdad?

- Una variable es una propiedad de un objeto interno especial, asociado con el bloque / función / script actualmente en ejecución.
- Trabajar con variables es realmente trabajar con las propiedades de ese objeto.

El entorno léxico es un objeto de especificación

"El entorno léxico "es un objeto de especificación: solo existe" teóricamente "en el language specification para describir cómo funcionan las cosas. No podemos obtener este objeto en nuestro código y manipularlo directamente.

Los motores de JavaScript también pueden optimizarlo, descartar variables que no se utilizan para ahorrar memoria y realizar otros trucos internos, siempre que el comportamiento visible permanezca como se describe.

Paso 2. Declaración de funciones

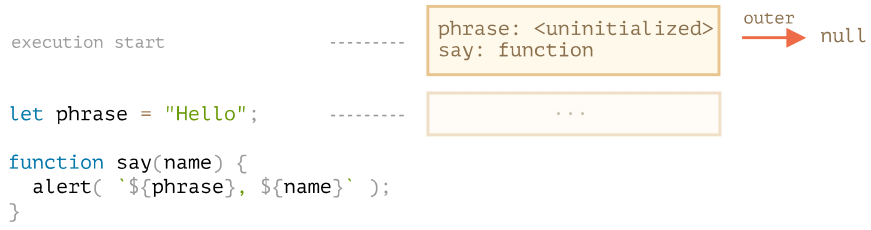
Una función también es un valor, como una variable.

La diferencia es que una `declaración de función` se inicializa completamente al instante.

Cuando se crea un entorno léxico, una declaración de función se convierte inmediatamente en una función lista para usar (a diferencia de `let`, que no se puede usar hasta la declaración).

Es por eso que podemos usar una función, declarada como `declaración de función`, incluso antes de la declaración misma.

Por ejemplo, aquí está el estado inicial del entorno léxico global cuando agregamos una función:

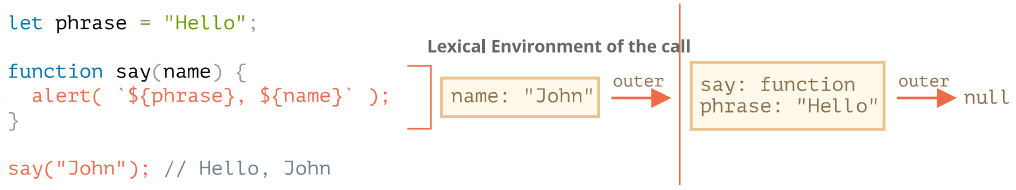


Naturalmente, este comportamiento solo se aplica a las `declaraciones de funciones`, no a las `expresiones de funciones`, donde asignamos una función a una variable, como `let say = function (name) ...`.

Paso 3. Entorno léxico interno y externo

Cuando se ejecuta una función, al comienzo de la llamada, se crea automáticamente un nuevo entorno léxico para almacenar variables y parámetros locales de la llamada.

Por ejemplo, para `say (" John ")`, se ve así (la ejecución está en la línea, etiquetada con una flecha):



Durante la llamada a la función tenemos dos entornos léxicos: el interno (para la llamada a la función) y el externo (global):

- El entorno léxico interno corresponde a la ejecución actual de `say`. Tiene una sola propiedad: `name`, el argumento de la función. Llamamos a `say("John")`, por lo que el valor de `name` es `"John"`.
- El entorno léxico externo es el entorno léxico global. Tiene la variable `phrase` y la función misma.

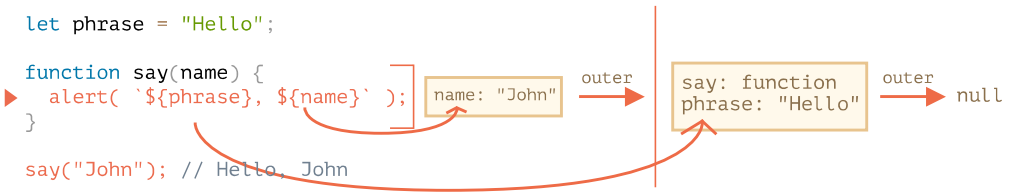
El entorno léxico interno tiene una referencia al `externo`.

Cuando el código quiere acceder a una variable: primero se busca el entorno léxico interno, luego el externo, luego el más externo y así sucesivamente hasta el global.

Si no se encuentra una variable en ninguna parte, se trata de un error en modo estricto (sin `use strict`, una asignación a una variable no existente crea una nueva variable global, por compatibilidad con el código antiguo).

En este ejemplo la búsqueda procede como sigue:

- Para la variable `name`, la `alert` dentro de `say` lo encuentra inmediatamente en el entorno léxico interno.
- Cuando quiere acceder a `phrase`, entonces no hay `phrase` localmente, por lo que sigue la referencia al entorno léxico externo y lo encuentra allí.



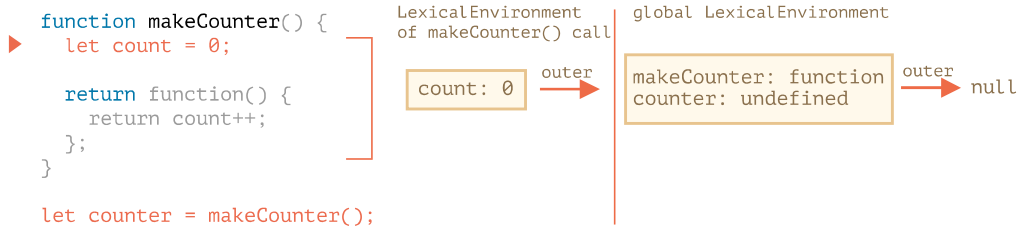
Paso 4. Devolviendo a function

Volvamos al ejemplo de `makeCounter`.

```
1  function makeCounter() {
2    let count = 0;
3
4    return function() {
5      return count++;
6    };
7  }
8
9  let counter = makeCounter();
```

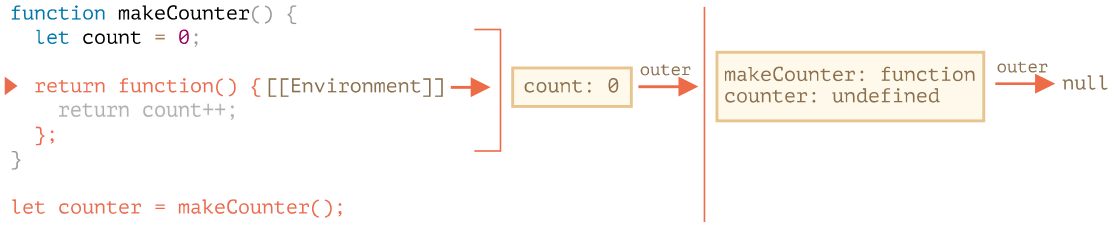
Al comienzo de cada llamada a `makeCounter()`, se crea un nuevo objeto de entorno léxico para almacenar variables para la ejecución `makeCounter`.

Entonces tenemos dos entornos léxicos anidados, como en el ejemplo anterior:



Lo que es diferente es que, durante la ejecución de `makeCounter()`, se crea una pequeña función anidada de solo una línea: `return count ++`. Aunque no la ejecutamos, solo la creamos.

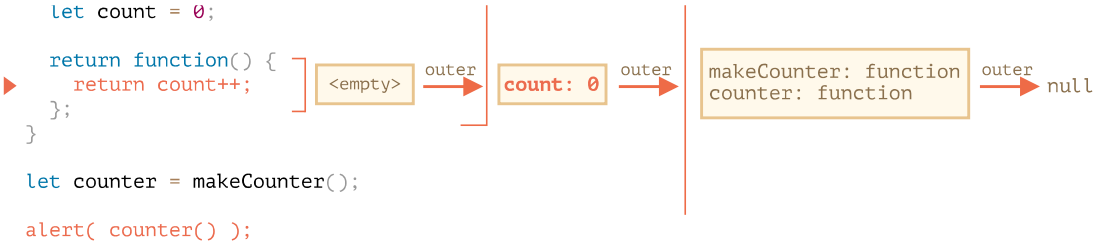
Todas las funciones recuerdan el entorno léxico en el que fueron realizadas. Técnicamente, no hay magia aquí: todas las funciones tienen la propiedad oculta llamada `[[Environment]]`, que mantiene la referencia al entorno léxico donde se creó la función:



Entonces, `counter.[[Environment]]` tiene la referencia a `{count: 0}` Entorno léxico. Así es como la función recuerda dónde se creó, sin importar dónde se llame. La referencia `[[Environment]]` se establece una vez y para siempre en el momento de creación de la función.

Luego, cuando `counter()` es llamado, un nuevo Entorno Léxico es creado por la llamada, y su referencia externa del entorno léxico se toma de `counter.[[Environment]]`:

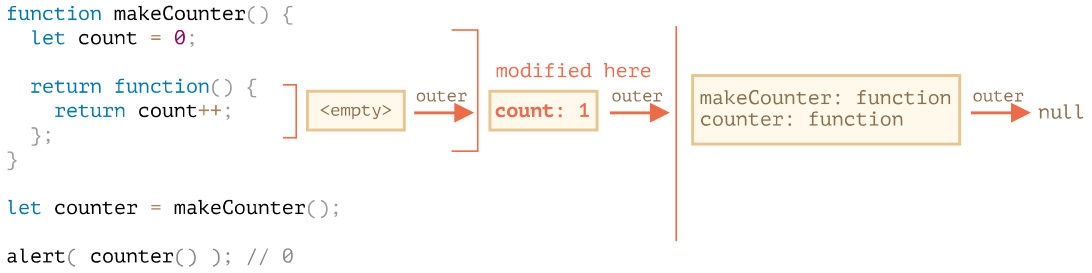
```
function makeCounter() {
```



Ahora cuando el código dentro de `counter()` busca `count` variable, primero busca su propio entorno léxico (vacío, ya que no hay variables locales allí), luego el entorno léxico del exterior llama a `makeCounter()`, donde lo encuentra y lo cambia.

Una variable se actualiza en el entorno léxico donde vive.

Aquí está el estado después de la ejecución:



Si llamamos a `counter()` varias veces, la variable `count` se incrementará a `2`, `3` y así sucesivamente, en el mismo lugar.

> **Cierre (Closure)**
>
> Existe un término general de programación "closure" que los desarrolladores generalmente deben conocer.
>
> Una clausura es una función que recuerda sus variables externas y puede acceder a ellas. En algunos lenguajes, eso no es posible, o una función debe escribirse de una manera especial para que suceda. Pero como se explicó anteriormente, en JavaScript, todas las funciones son clausuras naturales (solo hay una excepción, que se cubrirá en La sintaxis "new Function").
>
> Es decir: recuerdan automáticamente dónde se crearon utilizando una propiedad oculta `[[Environment]]`, y luego su código puede acceder a las variables externas.
>
> Cuando en una entrevista, un desarrollador frontend recibe una pregunta sobre "¿qué es una clausura?", Una respuesta válida sería una definición de clausura y una explicación de que todas las funciones en JavaScript son clausuras, y tal vez algunas palabras más sobre detalles técnicos: la propiedad `[[Environment]]` y cómo funcionan los entornos léxicos.

Recolector de basura

Por lo general, un entorno léxico se elimina de la memoria con todas las variables una vez que finaliza la llamada a la función. Eso es porque no hay referencias al respecto. Como cualquier objeto de JavaScript, solo se mantiene en la memoria mientras es accesible.

Sin embargo, si hay una función anidada a la que todavía se puede llegar después del final de una función, entonces tiene la propiedad `[[Environment]]` que hace referencia al entorno léxico.

En ese caso, el entorno léxico aún es accesible incluso después de completar la función, por lo que permanece vivo.

Por ejemplo:

```
1  function f() {
2    let value = 123;
3
4    return function() {
5      alert(value);
6    }
7  }
8
9  let g = f(); // g.[[Environment]] almacena una referencia al entorno léxico
10 // de la llamada f() correspondiente
```

Tenga en cuenta que si se llama a `f()` muchas veces y se guardan las funciones resultantes, todos los objetos del entorno léxico correspondientes también se conservarán en la memoria. Veamos las 3 funciones en el siguiente ejemplo:

```
function f() {
  let value = Math.random();

  return function() { alert(value); };
}

// 3 funciones en un array, cada una de ellas enlaza con el entorno léxico
// desde la ejecución f() correspondiente
let arr = [f(), f(), f()];
```

Un objeto de entorno léxico muere cuando se vuelve inalcanzable (como cualquier otro objeto). En otras palabras, existe solo mientras haya al menos una función anidada que haga referencia a ella.

En el siguiente código, después de eliminar la función anidada, su entorno léxico adjunto (y por lo tanto el `value`) se limpia de la memoria:

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // mientras exista la función g, el valor permanece en la memoria

g = null; // ... y ahora la memoria está limpia
```

Optimizaciones en la vida real

Como hemos visto, en teoría, mientras una función está viva, todas las variables externas también se conservan.

Pero en la práctica, los motores de JavaScript intentan optimizar eso. Analizan el uso de variables y si es obvio que el código no usa una variable externa, la elimina.

Un efecto secundario importante en V8 (Chrome, Edge, Opera) es que dicha variable no estará disponible en la depuración.

Intente ejecutar el siguiente ejemplo en Chrome con las Herramientas para desarrolladores abiertas.

Cuando se detiene, en el tipo de consola `alert(value)`.

```
function f() {
  let value = Math.random();

  function g() {
    debugger; // en console: type alert(value); ¡No hay tal variable!
  }

  return g;
}

let g = f();
g();
```

Como puede ver, ¡no existe tal variable! En teoría, debería ser accesible, pero el motor lo optimizó.

Eso puede conducir a problemas de depuración divertidos (si no son muy largos). Uno de ellos: podemos ver una variable externa con el mismo nombre en lugar de la esperada:

```
let value = "Surprise!";

function f() {
  let value = "the closest value";

  function g() {
    debugger; // en la consola escriba: alert(value); Surprise!
  }

```

```
  return g;
}

let g = f();
g();
```

Esta característica de V8 es bueno saberla. Si está depurando con Chrome/Edge/Opera, tarde o temprano lo encontrará.

Eso no es un error en el depurador, sino más bien una característica especial de V8. Tal vez en algún momento la cambiarán. Siempre puede verificarlo ejecutando los ejemplos en esta página.

Tareas

¿Una función recoge los últimos cambios?

importancia: 5

La función sayHi usa un nombre de variable externo. Cuando se ejecuta la función, ¿qué valor va a utilizar?

```
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

sayHi(); // ¿qué mostrará: "John" o "Pete"?
```

Tales situaciones son comunes tanto en el desarrollo del navegador como del lado del servidor. Se puede programar que una función se ejecute más tarde de lo que se creó, por ejemplo, después de una acción del usuario o una solicitud de red.

Entonces, la pregunta es: ¿recoge los últimos cambios?

solución

¿Qué variables están disponibles?

importancia: 5

La función `makeWorker` a continuación crea otra función y la devuelve. Esa nueva función se puede llamar desde otro lugar.

¿Tendrá acceso a las variables externas desde su lugar de creación, o desde el lugar de invocación, o ambos?

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
}

let name = "John";

// crea una función
let work = makeWorker();

// la llama
work(); // ¿qué mostrará?
```

¿Qué valor mostrará? "Pete" o "John"?

solución

¿Son independientes los contadores?

importancia: 5

Aquí hacemos dos contadores: `counter` y `counter2` usando la misma función `makeCounter`.

¿Son independientes? ¿Qué va a mostrar el segundo contador? `0,1` o `2,3` o algo más?

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
let counter2 = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1

alert( counter2() ); // ?
alert( counter2() ); // ?
```

solución

Objeto contador

importancia: 5

Aquí se crea un objeto contador con la ayuda de la función constructora.

¿Funcionará? ¿Qué mostrará?

```
function Counter() {
  let count = 0;

  this.up = function() {
    return ++count;
  };
  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // ?
alert( counter.up() ); // ?
alert( counter.down() ); // ?
```

solución

Función en if

Mira el código ¿Cuál será el resultado de la llamada en la última línea?

```
let phrase = "Hello";

if (true) {
  let user = "John";

  function sayHi() {
    alert(`${phrase}, ${user}`);
  }
}

sayHi();
```

solución

Suma con clausuras

importancia: 4

Escriba la función `sum` que funcione así: `sum(a)(b) = a+b` .

Sí, exactamente de esta manera, usando paréntesis dobles (no es un error de tipeo).

Por ejemplo:

```
sum(1)(2) = 3
sum(5)(-1) = 4
```

solución

¿Es visible la variable?

importancia: 4

¿Cuál será el resultado de este código?

```
let x = 1;

function func() {
  console.log(x); // ?

  let x = 2;
}

func();
```

P.D Hay una trampa en esta tarea. La solución no es obvia.

solución

Filtrar a través de una función

importancia: 5

Tenemos un método incorporado `arr.filter(f)` para arrays. Filtra todos los elementos a través de la función `f` . Si devuelve `true` , entonces ese elemento se devuelve en el array resultante.

Haga un conjunto de filtros "listos para usar":

- `inBetween(a, b)` – entre `a` y `b` o igual a ellos (inclusive).
- `inArray([...])` – en el array dado

El uso debe ser así:

- `arr.filter(inBetween(3,6))` – selecciona solo valores entre 3 y 6.
- `arr.filter(inArray([1,2,3]))` – selecciona solo elementos que coinciden con uno de los miembros de `[1,2,3]` .

Por ejemplo:

```
/* .. tu código para inBetween y inArray */

let arr = [1, 2, 3, 4, 5, 6, 7];

alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6

alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Abrir en entorno controlado con pruebas.

solución

Filtrar inBetween

Filtrar inArray

Ordenar por campo

importancia: 5

Tenemos una variedad de objetos para ordenar:

```
let users = [
  { name: "John", age: 20, surname: "Johnson" },
  { name: "Pete", age: 18, surname: "Peterson" },
  { name: "Ann", age: 19, surname: "Hathaway" }
];
```

La forma habitual de hacerlo sería:

```
// por nombre(Ann, John, Pete)
users.sort((a, b) => a.name > b.name ? 1 : -1);

// por edad (Pete, Ann, John)
users.sort((a, b) => a.age > b.age ? 1 : -1);
```

¿Podemos hacerlo aún menos detallado, como este?

```
users.sort(byField('name'));
users.sort(byField('age'));
```

Entonces, en lugar de escribir una función, simplemente ponga `byField(fieldName)`.

Escriba la función `byField` que se pueda usar para eso.

Abrir en entorno controlado con pruebas.

solución

Ejército de funciones

importancia: 5

El siguiente código crea una serie de `shooters`.

Cada función está destinada a generar su número. Pero algo anda mal ...

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {
    let shooter = function() { // crea la función shooter
      alert( i ); // debería mostrar su número
    };
    shooters.push(shooter); // y agregarlo al array
    i++;
  }

  // ...y  devolver el array de tiradores
  return shooters;
}

let army = makeArmy();
```

```
18
19  // ... todos los tiradores muestran 10 en lugar de sus 0, 1, 2, 3 ...
20  army[0](); // 10 del tirador número 0
21  army[1](); // 10 del tirador número 1
22  army[2](); // 10 ...y así sucesivamente.
```

¿Por qué todos los tiradores muestran el mismo valor?

Arregle el código para que funcionen según lo previsto.

Abrir en entorno controlado con pruebas.

solución

Lección anterior Próxima lección

Compartir Mapa del Tutorial

Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor enviar una propuesta de GitHub o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>`, para varias líneas – envolverlas en la etiqueta `<pre>`, para más de 10 líneas – utilice una entorno controlado (sandbox) (plnkr, jsbin, codepen...)

0 Comentarios es.javascript.info Política de privacidad de Disqus Acceder

Recomendar 1 Tweet Compartir Ordenar por los mejores

Sé el primero en comentar...

INICIAR SESIÓN CON O REGISTRARSE CON DISQUS ?

Nombre

Sé el primero en comentar.

Suscríbete Añade Disqus a tu sitio webAñade Disqus Añadir Do Not Sell My Data

© 2007—2021 Ilya Kantoracerca del proyecto contáctenos