ACÁMICA

TEMA DEL DÍA

Funciones de orden superior

Son funciones que operan con otras funciones, bien sea tomándolas como parámetros, o retornándolas.

Agenda

Daily

Abstracciones

Funciones de orden superior

Actividades

Cierre

Daily

Daily

Sincronizando...

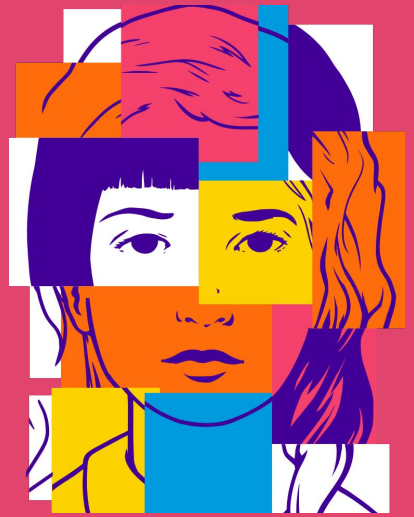
Toolbox

¿Cómo te ha ido?
¿Obstáculos?
¿Cómo seguimos?

Challenge

¿Cómo te ha ido?
¿Obstáculos?
¿Cómo seguimos?

Funciones de orden superior



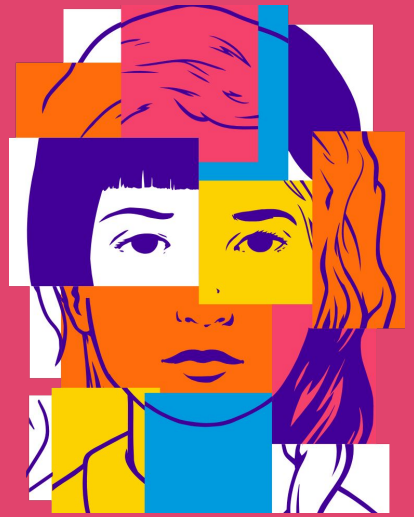
Funciones de orden superior

Son funciones que toman como argumento otras funciones.

En la meeting de hoy vamos a aprender a usar 2 de estas funciones, las cuales forman parte de Javascript y se aplican directamente sobre arrays.

```
.map()
.filter()
```

`.map()`



.map()

Itera sobre una colección de elementos permitiendo aplicar una transformación sobre cada uno.

```
const lista = [1,2,3];

const listaAlCuadrado = lista.map(function (elemento) {
  return elemento * elemento;
})
// 1, 4, 9
```

Programamos

Conozcamos .map()

La función .map() es una FOS (función de orden superior) que nos permite iterar de manera sencilla por un array. Su ventaja, o superpoder, es que puede crear un array nuevo, con elementos de otro array, transformados.

Necesitamos a una persona voluntaria que haga el siguiente paso a paso. En un sandbox nuevo, o entorno de Javascript, codea el siguiente array:

```
let numeros = [1,2,3,4,5,6];
```

Y vamos a asumir la siguiente consigna: necesitamos crear un array nuevo, pero donde los números están multiplicados por 2.

Conozcamos .map()

Para esto, podemos utilizar .map(). Para aplicar esta FOS, tenemos que invocarla sobre el array original, sobre el cual queremos iterar:

```
numeros.map()
```

Pero, como te mencionamos, las FOS toman como parámetro otras funciones. Por ejemplo, supongamos que tenemos una función que retorna el doble de un número:

```
function calcularDoble(numero) {
  return numero * 2;
}
```

Podemos tomar esta función y pasarla a .map(), de esta forma:

```
numeros.map(calcularDoble)
```

Finalmente, como te dijimos que esta FOS nos permite retornar un nuevo array, podemos finalmente escribir lo siguiente:

```
let nuevosNumeros = numeros.map(calcularDoble)
```

¿Que hace .map() ?

```
let nuevosNumeros = numeros.map(calcularDoble)
```

Esta FOS itera por cada uno de los items originales del array “numeros” y le aplica la función “calcularDoble” a cada uno. El resultado retornado se pasa como un nuevo item a “nuevosNumeros”, el cual tendrá el siguiente valor:

```
console.log(nuevosNumeros);
// [2,4,6,8,10,12]
```

Ejercicio

Necesitamos a una nueva persona voluntaria para resolver este ejercicio.
Partiendo de este array:

```
let numeros = [1,2,3,4,5,6];
```

Se desea obtener un nuevo array que tenga los números del array anterior, pero cada item debe ser sumado con el número 10, es decir, se tendrá que obtener el array:

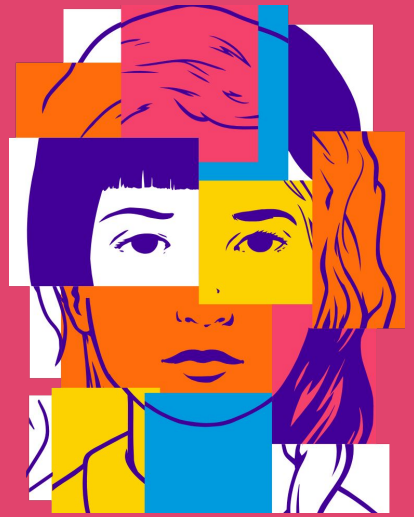
```
[11,12,13,14,15,16]
```

Utiliza .map() para esto.

¡BREAK!

Ph. Credit: Drew Coffmann

.filter()



.filter()

Crea un nuevo array con los elementos que pasen una condición.

```
let palabras = ['limite', 'elite', 'exhuberante',];

let resultado = palabras.filter(function(palabra) {
  return palabra.length <= 6
});

console.log(resultado);
['limite', 'elite'];
```

Programamos

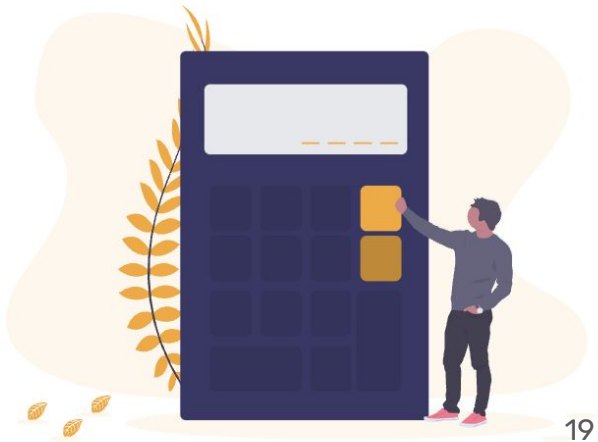
Conozcamos .filter()

La función .filter() es una FOS (función de orden superior) que nos permite iterar de manera sencilla por un array. Su ventaja, o superpoder, es que puede crear un array nuevo con elementos que pasen una condición dada.

Necesitamos a una persona voluntaria que haga el siguiente paso a paso. En un sandbox nuevo, o entorno de Javascript, codea el siguiente array:

```
let numeros = [1,2,3,4,5,6];
```

Y vamos a asumir la siguiente consigna: necesitamos crear un array nuevo, pero donde los números sean sólo mayor a 3.



Conozcamos .filter()

Para esto, podemos utilizar .map(). Para aplicar esta FOS, tenemos que invocarla sobre el array original, sobre el cual queremos iterar:

```
numeros.filter();
```

Pero, como te mencionamos, las FOS toman como parámetro otras funciones. Por ejemplo, supongamos que tenemos una función que retorna el doble de un número:

```
function esMayorAtres(numero) {
  return numero > 3;
}
```

Podemos tomar esta función y pasarla a .map(), de esta forma:

```
numeros.filter(esMayorAtres)
```

Finalmente, como te dijimos que esta FOS nos permite retornar un nuevo array, podemos escribir lo siguiente:

```
let nuevosNumeros = numeros.filter(esMayorAtres)
```

Ejercicio

Necesitamos a una nueva persona voluntaria para resolver este ejercicio.
Partiendo de este array:

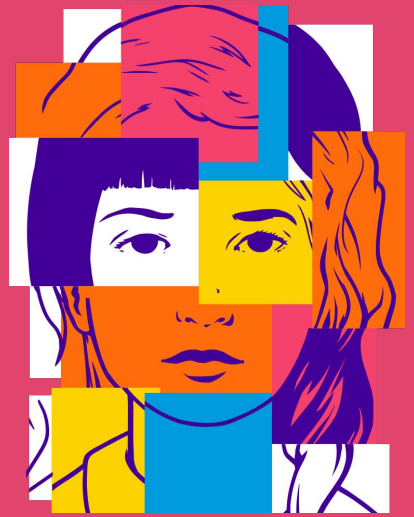
```
let numeros = [1,2,3,4,5,6];
```

Se desea obtener un nuevo array que tenga los números del array anterior, pero la condición es que los números sean solo pares. Es decir, el resultado tendría que ser:

```
[2,4,6];
```

Utiliza .filter() para esto.

FOS en colecciones



.map() en colecciones

Estas dos funciones, map y filter, nos ayudan a operar en colecciones también! Veamos cómo podemos hacer una nueva colección, con propiedades específicas, utilizando map.

Supongamos que tenemos esta colección:

```
let personas = [
  {
    nombre: "Carolina",
    edad: 20
  },
  {
    nombre: "Daniel",
    edad: 30
  },
  {
    nombre: "Luis",
    edad: 15
  }
]
```

Y queremos producir una nueva colección, solo con los nombres de las personas:

```
let personas = [
  {
    nombre: "Carolina",
  },
  {
    nombre: "Daniel",
  },
  {
    nombre: "Luis",
  }
]
```

Programamos

Paso 1: seteo de entorno

Necesitamos una persona voluntaria que empiece el ejercicio. Abre un code sandbox, o entorno de Javascript, y codea la colección de personas.

Vamos a utilizar la FOS map para crear una nueva colección de personas, donde cada objeto tenga solo la propiedad “nombre”.

```
let personas = [
  {
    nombre: "Carolina",
    edad: 20
  },
  {
    nombre: "Daniel",
    edad: 30
  },
  {
    nombre: "Luis",
    edad: 15
  }
]
```

Paso 2: implementar el map

Como la función .map() tiene la habilidad de retornar un array nuevo, escribamos una declaración que nos permita obtenerlo:

```
let soloNombres = personas.map();
```

El primer argumento que recibe .map() es una función, que se aplicará en cada item de la colección. ¿Sabias que podemos crear funciones anónimas, directamente en la lista de argumentos de la función?

```
// función anónima declarada
let soloNombres = personas.map(function(item) {

});
```

```
// función anónima arrow, o flecha
let soloNombres = personas.map((item) => {

});
```

Paso 3: definir la función

La función que recibe .map como argumento, es una función que se ejecutará en cada item del array original. Este “item” deberá ser especificado como argumento de esta función interna:

```
let soloNombres = personas.map((item) => {
  console.log(item) // ¿quién es item?
});
```

¿Qué representa “item” dentro de esta función? Escribe el código anterior y mira la consola.

Tip:

La clave en entender .map es la siguiente: el nuevo array que retornara, será compuesto de ítems retornados por la función anónima.

Es decir, si el return de la función anónima fuera el número 4:

```
let soloNombres = personas.map((item) => {
  return 4;
});
```

El nuevo array, “soloNombres”, tendrá el número “4”, tres veces.

```
console.log(soloNombres) // [4,4,4]
```

¿Por que tres veces? Por que esta función se ejecuta tantas veces como ítems tenga el array original, en este caso, “personas”.

Paso 4: return

Dicho esto, si queremos construir un array nuevo con objetos que solo tengan la propiedad “nombre”, tendremos que hacer esto:

```
let soloNombres = personas.map((item) => {
  return {
    nombre: item.nombre
  }
});
```

Al hacer esto, el array “soloNombres” tendrá 3 objetos, con la propiedad “nombre” cada uno, pero el valor de nombre será correspondiente al valor del nombre de cada item del array original.

.filter() en colecciones

Con .filter() podemos aplicar una lógica similar.

Supongamos que tenemos esta colección:

```
let personas = [
  {
    nombre: "Carolina",
    edad: 20
  },
  {
    nombre: "Daniel",
    edad: 30
  },
  {
    nombre: "Luis",
    edad: 15
  }
]
```

Y queremos producir una nueva colección, donde las personas sean mayores de 15 años.

```
let personas = [
  {
    nombre: "Carolina",
    edad: 20
  },
  {
    nombre: "Daniel",
    edad: 30
  }
];
```

Programamos

Paso 1: implementación

A diferencia de .map, .filter retorna un item al array nuevo, según una condición dada.

Partiendo de la misma colección de personas, escribe la siguiente implementación:

```
let mayoresDeQuince = personas.filter()
```

Paso 2: función

El nombre del “item” puede ser lo que queramos, por ejemplo, “persona.

Escribe la estructura básica de la función interna, de esta manera:

```
let mayoresDeQuince = personas.filter((persona) => {


});
```

Paso 3: condición

.filter() le va a pasar a su nuevo array, “mayoresDeVeinte”, todos los objetos que cumplan una condición, en nuestro caso, si la edad de la persona es mayor a 15 años.

```
let mayoresDeQuince = personas.filter((persona) => {
  if(persona.edad > 15) {
    return true;
  }
})
```

Si la función anónima de .filter retorna “true”, el objeto es pasado al nuevo array.

Para la casa

Práctica

INVENTORXS 💡

Partiendo de este sandbox, realicemos las consignas dadas mientras practicamos las funciones de alto nivel previamente explicadas.

Práctica

Toolbox challenge refactor

Partiendo de lo realizado en el challenge de la toolbox, refactoriza los ejercicios para cumplir con las siguientes consignas:

Práctica

Personas asistentas

Utiliza .filter para crear un nuevo array con las personas asistentes, es decir, que tengan la propiedad “asistente” dentro de su objeto, y que esta sea igual a “true”.

Práctica

Solo personas

Utiliza .map para crear un nuevo array que contenga solo el nombre de las personas.

Para la próxima

1) Termina los ejercicios de la meeting de hoy.
2) Lee la toolbox 24.
3) Resuelve el challenge.

ACÁMICA