

El camaleón de Javascript

"Hay dos formas de construir un diseño de software, una es hacerlo tan simple y que obviamente no existan deficiencias, y otra es hacerlo tan complicado que no existan deficiencias obvias"

C.A.R. Hoare, 1980 – ACM Turing Award Lecture

Durante esta primera parte del sprint nos hemos dedicado a estudiar los tipos de valores que ofrece Javascript como lenguaje. Hemos puesto en práctica la sintaxis de cómo declarar [variables](#), [condicionales](#), [ciclos](#) y [funciones](#), pudiendo mezclar conceptos para desarrollar funcionalidades complejas.

En la toolbox de hoy vamos a profundizar un poco más el concepto de la función, tratándola como un **valor** más del lenguaje, lo que nos va a permitir realizar nuevas operaciones con ella, extendiendo así el comportamiento típico que hemos visto hasta ahora. Esto nos va a llevar al concepto de [funciones de orden superior](#), las cuales son aquellas funciones que operan sobre otras funciones, bien sea pasándolas como argumentos, o siendo retornadas dentro de otras.

Pero, antes de esto, vamos a completar el mapa de los valores en Javascript. Agregaremos un par de ítems a la categoría de valores primitivos: ``null`` y ``undefined``.

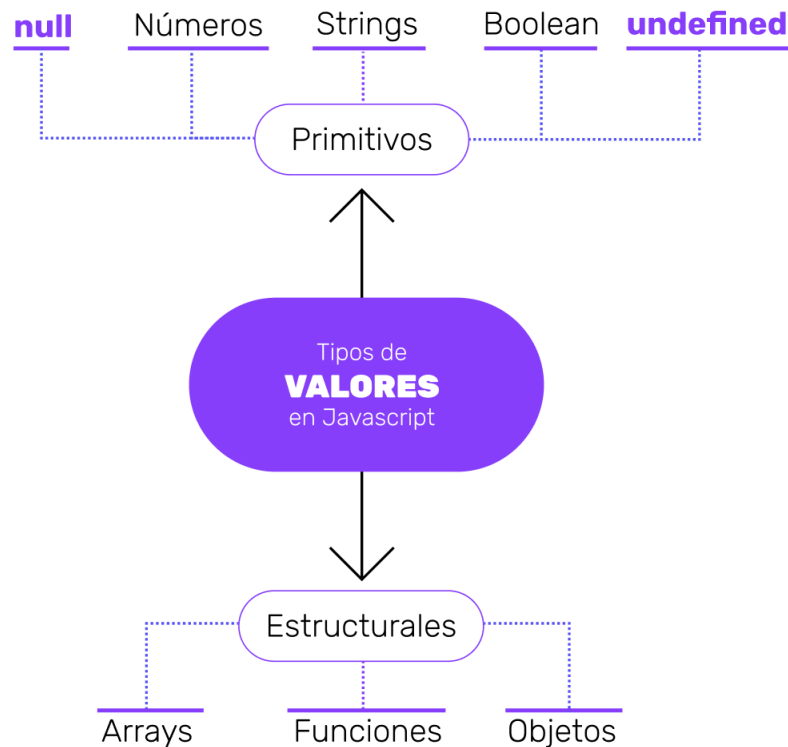
Valores primitivos: `null` y `undefined`

Como ya sabemos, los tipos de valores en Javascript pueden dividirse en dos categorías: [primitivos](#) y [estructurales](#).

Los [primitivos](#) son considerados los "átomos" del lenguaje, y ofrecen la posibilidad de crear datos aislados, mientras que los [estructurales](#) son aquellos que nos permiten conglomerar varios datos primitivos en una variable, construyendo "moléculas".

Para completar la categoría de datos primitivos, te contamos sobre ``null`` y ``undefined``, los cuales actúan como valores "vacíos" y "no definidos" respectivamente.





El valor `null` es un valor definido, pero representa un valor vacío. Este puede ser usado cuando se declara una variable sobre la cual no sabemos su valor de inicio, por ejemplo:

```
...
```

```
let dato = null;
```

```
console.log(dato) // null
```

```
dato = 43;
```

```
console.log(dato) // 43
```

```
...
```

Mientras que `undefined` es el valor, por defecto, para los casos donde una variable ha sido declarada, pero no tiene ningún valor.

```
...
```

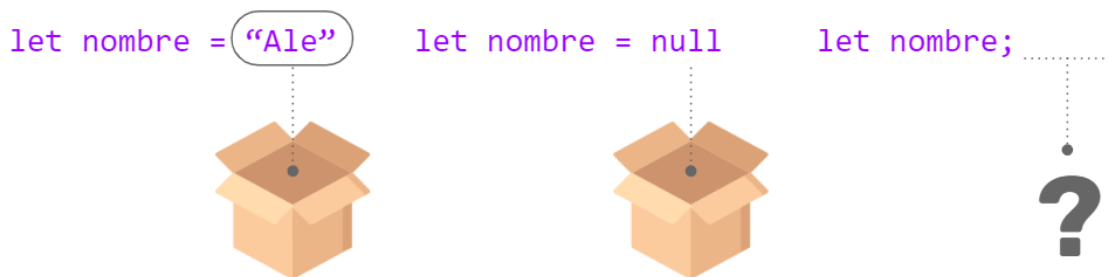
```
let dato;
```

```
console.log(dato) // undefined
```

```
...
```



Lo anterior significa que ``null`` sirve para declarar **un valor vacío**, mientras ``undefined`` se usa cuando el valor no ha sido declarado, o no definido. Si imaginamos las variables como cajas contenedoras de un valor, podríamos entonces crear esta representación gráfica:



En el siguiente video podrás ver un poco más las diferencias entre estos dos valores, así como un caso de uso.

[El caso de null y undefined](#)

Valores estructurales: función

¿Cómo definirías el concepto de la función en Javascript? Hasta ahora, hemos utilizado la función como un contenedor de instrucciones, o pasos a ejecutar, el cual toma opcionalmente parámetros de entrada para realizar un proceso y emitir una salida, o resultado. Esencialmente, una función es eso, algo que toma uno o varios datos y los procesa para producir algo nuevo. Toma el ejemplo de una función que suma dos números:

...

```
function suma(x,y) {
    return x + y;
}
```

```
suma(2,2) // retorna 4
```

...

Pero, como habrás visto en el gráfico que te mostramos al principio, las



funciones están dentro de la categoría de **valores estructurales**. El tipo de función presentado en el ejemplo de la suma se llama “**función declarada**”, y consiste en crear una función con la sintaxis que ya conoces, a través de la palabra reservada ``function`` junto al nombre, los paréntesis de los parámetros y el cuerpo entre llaves.

Te preguntarán, ¿cómo es posible que la función sea un tipo de valor, si no lo declaramos con la misma sintaxis que un array, un string, o un número?

La respuesta radica en el hecho de que una función puede ser creada de varias formas, inclusive como un valor nuevo asignado a una variable. Presta atención al siguiente ejemplo:

```
...  
  
let suma = function(x,y) {  
    return x+y;  
};  
  
let resultado = suma(2,2);  
  
console.log(resultado); // 4  
  
...
```

Observa cómo creamos una función usando una sintaxis similar a como cuando queremos crear una variable primitiva: usamos la palabra ``let`` (podría ser ``const`` también) seguida por el nombre de la variable, cuyo valor va a ser igual a una función anónima, la cual podrá ser invocada con el nombre declarado a la variable. ¿Recuerdas cuando le pasamos una función anónima a una propiedad dentro de un objeto?

```
...  
  
let persona = {  
    nombre: "Johana",  
    saluda: function() {  
        console.log("hola soy " + this.nombre)  
    }  
}
```



...

Estamos haciendo algo similar, pero fuera de un objeto. Estamos creando una variable de tipo ``function`` cuyo valor es exactamente eso, una función. Este tipo de creación se llama “**función expresada**”. Existen otras formas de crear funciones, las cuales estaremos viendo más adelante en el programa. Si quieres saltar un poco al futuro, te sugerimos leer [este artículo](#) donde se describen las distintas formas de declarar funciones en Javascript.

Entidades de primera clase

Al ser un tipo de valor, las funciones pueden comportarse como otros valores (números, strings, arrays, etc). Con estos hemos operado de distintas formas durante nuestras prácticas, bien sea retornándolos dentro de una función, o pasándolos como parámetros.

Las funciones de Javascript también tienen estas mismas capacidades. Son camaleónicas, por decirlo de alguna forma, ya que pueden operar como cualquier otro valor que hemos visto hasta ahora. En programación, cuando una entidad soporta las operaciones disponibles que aplican sobre el resto de sus valores (números, strings, booleanos, etc), se le conoce como una **entidad de primera clase** o *first class citizen*. Para el caso de Javascript, las funciones se consideran “first class functions”, o [funciones de primera clase](#).

Estas son algunas de las operaciones permitidas que tienen las funciones dentro de Javascript:



Pasarlas a otras
funciones



Almacenarlas
en un array



Declararlas como
valores en un objeto



Crearlas como
variables



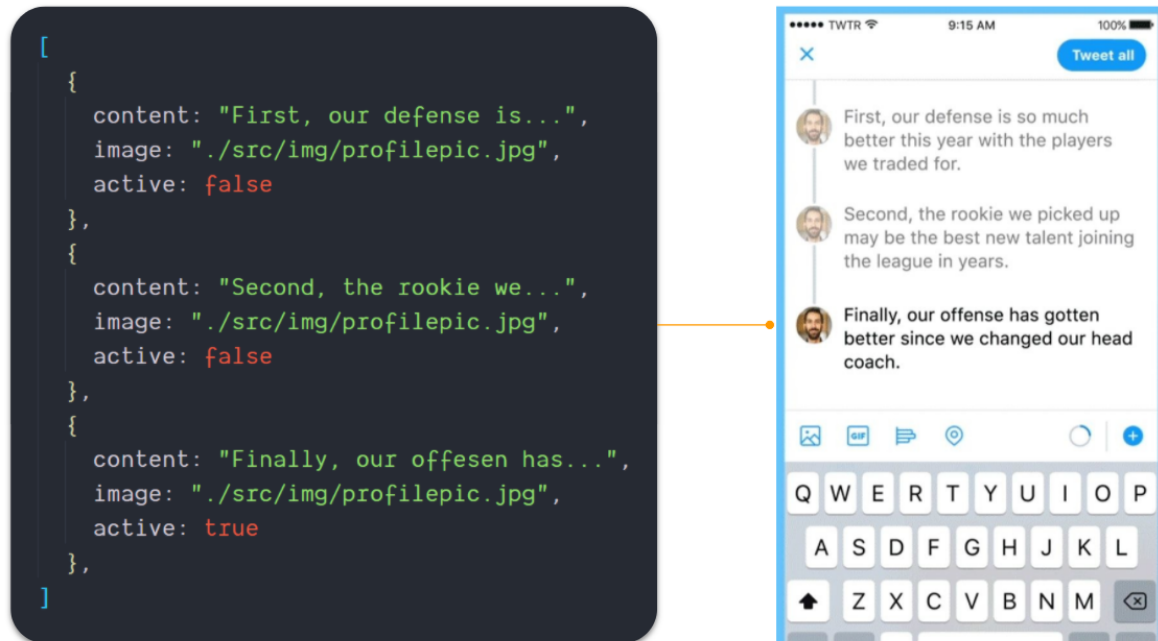
Retornarlas dentro
de una función

En esta toolbox queremos contarte específicamente sobre una de estas propiedades: **pasar funciones como argumentos a otras funciones** para ser ejecutadas, o llamadas después, pero dentro de la función que la recibe como argumento. A este concepto lo conocemos como un [callback](#).



Callbacks

Repetir instrucciones en programación es una actividad muy común. Listas de personas usuarias, tweets, posts, o comentarios son ejemplos de estructuras de datos las cuales tendrían que ser iteradas para poder mostrarlas en una interfaz gráfica. En el siguiente ejemplo verás cómo un array de objetos sirve como estructura de datos para mostrar una lista de tweets, en el cual tendremos que iterar para poder construir la interfaz gráfica.



Un callback es la función que se pasa como argumento a otra función la cual será ejecutada en otro momento. Para ejemplificar este concepto, veamos el siguiente código:

```
...
```

```

for (let i=0; i <= 9; i++) {
    console.log(i)
}

```

```
...
```

Esta aplicación imprime por consola los números del 0 al 9, utilizando un ciclo `for`.



En este ejemplo, estamos especificando la acción (el `console.log`) dentro del ciclo, pero si repetir instrucciones es algo tan común, quizás deberíamos plantear una forma más extensible de ejecutarlas. Por cierto, ¿sabías que ``console`` es un objeto con muchas utilidades, y ``log``, una de sus funciones? Explora más de sus utilidades en [este artículo](#).

Entonces, nuestro objetivo es crear una forma más extensible para iterar una N cantidad de veces, en vez de escribir un ciclo cada vez que necesitemos repetir algo. Una primera aproximación puede ser de esta forma:

...

```
function iterar(n) {  
    for (let i=0; i <= n; i++) {  
        console.log(i)  
    }  
}
```

...

De esta manera podemos parametrizar hasta dónde llegaría el ciclo, así como abstraer el comportamiento del mismo en una función. Con esta opción no tenemos que escribir más el ciclo, sino sólo invocar la función ``iterar``. Sin embargo, esta función está fuertemente vinculada a la idea de mostrar un mensaje por consola, nos sirve solo para eso, haciéndola poca extensible. En el siguiente video te proponemos una forma más independiente de ejecutar una acción al pasar una función como argumento a ``iterar()``.

[Pasar funciones a otras funciones](#)

Tampoco es necesario que definamos previamente la función y pasarla como un callback, ya que también podemos definir directamente la función en la lista de argumentos:

...

```
let etiquetas = [];
```



```
iterar(5, function(){  
    etiquetas.push("Etiqueta " + i);  
});  
...
```

El siguiente vídeo profundiza sobre esta idea, ¡dale un vistazo!

[Pasando funciones como argumentos](#)

Funciones de orden superior

Todo lo anterior nos trae al concepto de funciones de orden superior, ya que éstas son funciones que operan en otras funciones, bien sea tomándolas como argumentos, como vimos en esta toolbox, o siendo retornadas por otras funciones, como veremos más adelante.

Como hemos visto que las funciones son simplemente un valor más dentro de Javascript, no hay nada realmente remarcable que debamos aportar a este concepto. El término de "función de orden superior" (o *high order function* en inglés) viene de las matemáticas, donde el concepto de funciones y otros valores es mucho más acentuado.

Dentro de Javascript, tenemos disponibles una serie de funciones de orden superior, las cuales operan exclusivamente sobre arrays. Estas funciones son:


- map
- filter
- find
- forEach


En la próxima meeting vamos a ver los casos de uso más comunes donde estas funciones de orden superior operan, aplicándolas directamente en los ejercicios que hemos venido construyendo.


¡Prepárate para la próxima meeting!



Profundiza


 [Hablemos de callbacks](#). Este artículo nos da otro punto de vista sobre cómo emplear callbacks en código.

 En la toolbox te comentamos cómo las funciones pueden ser también retornadas dentro de otras funciones. En [este video](#) se construye un ejemplo más complejo evidenciando esto, a través del caso de uso de una calculadora.

 En la próxima meeting estaremos hablando sobre algunas funciones de orden superior que aplican sobre los arrays. Si quieres empezar a indagar más sobre éstas, te dejamos la documentación oficial de MDN para cada una:


- [map](#)
- [filter](#)
- [find](#)
- [forEach](#)


Comunidad


 ¿Qué dicen lxs developers sobre las callbacks? Explora [este hilo en Quora](#), donde varias personas dan su punto de vista sobre este concepto.

Challenges

 1. Sigue las instrucciones de [este sandbox](#) y calcula la cantidad de personas asistentes a un evento.

 2. Sobre el sandbox anterior, desarrolla una función que te permita crear un nuevo array que tenga solo las personas asistentes. Es decir, que tengan la propiedad "asistente" dentro de su objeto, y que esta sea igual a "true".

 3. Además, desarrolla una función que te permite crear un nuevo array que tenga solo las personas de la sección "platino".

 4. Finalmente, crea una única función que cree un nuevo array según un filtro y un valor especificado, los cuales tendrán que ser pasados como argumentos a la función. Por ejemplo:



```
let speakers = filtrarPersonas("rol", "speaker");
```

Lo anterior deberá retornar un nuevo array solo con los objetos donde la propiedad "rol" sea igual a "speaker. Encuentra unos tipos en [este sandbox](#).

