clase # 4

¿Cómo llega el script al navegador?

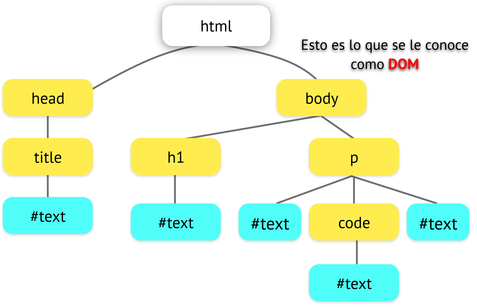
DOM

document object model

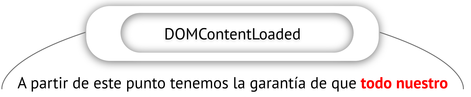
Es la representación que hace el **navegador** de un documento **HTML**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Scope</title>
</head>
<body>
    <h1>DOM</h1>
    <p>
        Cuando llega el HTML al browser, este lo empieza a pasar:
        va leyendo etiqueta por etiqueta y va creando el DOM.
        Cuando este proceso termina por completo es cuando
        obtenemos el evento <code>DOMContentLoaded</code>
    </p>
</body>
</html>
```

Cuando el navegador recibe este archivo **lo tiene que convertir** en una estructura parecida **a un árbol**



Cuando termina el **navegador** de convertirlo al **DOM** ocurre el **evento:**



A partir de este punto tenemos la garantía de que **todo nuestro documento se ha cargado**.

SCRIPT EXTERNOS O EMBEBIDOS

Todo script que carguemos en nuestra página tiene un llamado y una ejecución.

cuando el DOM se este procesando va a detener todo el procesamiento cuando se encuentre la etiqueta **<script>**

```
<body>
    <script>
        Estos script pueden ser externos o embebidos.
    </script>

    Cuando esto ocurre no se va a leer ningún otro elemento
    HTML, hasta que acabemos con el script.

    <p>
        Soy un párrafo, que no se puede ejecutar
        todavía por culpa del script de arriba 😠
    </p>
</body>
```

por lo tanto, es importante en que lugar colocamos nuestros **sciprt's.**

```
<body>
    <h1> titulo </h1>
    <p>  texto  </p>
    <script>El mejor lugar para colocarlo es al final del body <script>
</body>
```

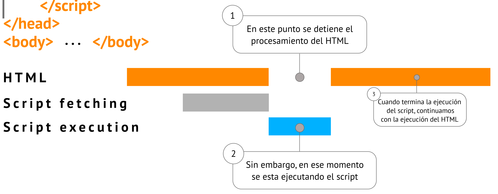
ATRIBUTO ASYNC Y DEFER

Tanto con async como defer podemos hacer **llamadas asíncronas** pero tienen sus diferencias:

ASYNC:

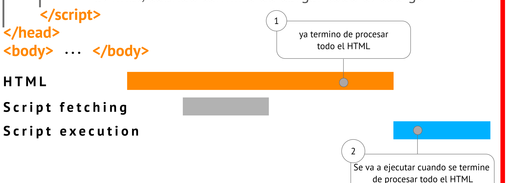
Estos atributos aplican para los script externos

```
<head>
    <script async  src="url">
        Con async podemos hacer la petición de forma asíncrona y
        no vamos a detener la carga del DOM hasta que se haga la
        ejecución del código
    </script>
</head>
<body>  ...  </body>
```



DEFER:

```
<head>
    <script defer  src="url">
        La petición es igual asíncrona como en el async pero va a
        deferir la ejecución del JavaScript, es decir se va a ejecutar
        al final, cuando termine de cargar todo el código HTML.
    </script>
</head>
<body>  ...  </body>
```



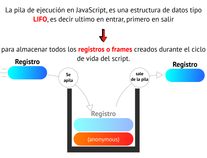
NOTAS

Pila de ejecución

CallStack

La pila de ejecución en JavaScript, es una estructura de datos tipo **LIFO**, es decir ultimo en entrar, primero en salir

para almacenar todos los **registros o frames** creados durante el ciclo de vida del script.



¿Registro?

Muchas veces se dice que lo que se guarda en la pila de ejecución son las funciones o los llamados a las funciones

Pero como vimos, en verdad lo que se apila es un **frame o registro** que contiene **información** de la función que se llamo.

Este registro va a **contener toda la información** necesaria para el que el motor pueda ejecutar esta función.

y ante un nuevo llamado a una función

Esto es fundamental para entender como funciona JavaScript, ya que:

El motor puede ejecutar **una** sola cosa a la vez

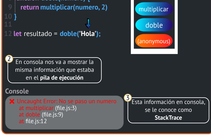
¿Qué es (anonymous)?

cuando ejecutamos un programa en JavaScript, la primara función que se agrega a la pila es una función **anónima** que engloba a todo el programa

Significa que se **termino la ejecución** del programa principal

Esta función está presente de forma **predeterminada en la pila de ejecución** y está en la parte inferior de la pila

¿Qué pasa si ocurre un error?



¿Qué es StackTrace?

Es la **secuencia** de llamadas que se fueron dando durante la ejecución de un programa hasta que **sucedió una excepción** o un error

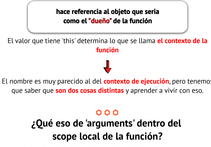
¿Qué información contiene cada uno de estos registro?

El nombre de la función, el archivo al que pertenece, el numero de la próxima línea a ejecutar y su **scope**

¿Qué es el scope?

Es el contexto actual de ejecución

El contexto actual de ejecución **de una función**, esta determinado por el **conjunto** de variables, parámetros y funciones que se puede **acceder** desde esa función **cuando se esta ejecutando.**



¿Qué eso de 'this' dentro del scope local de la función?

this, es una variable muy especial que **tienen** las funciones de JavaScript.

hace referencia al objeto que sería como el "dueño" de la función

El valor que tiene 'this' determina lo que se llama **el contexto de la función**

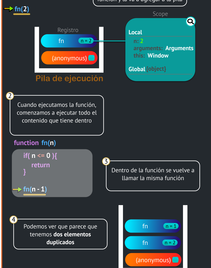
El nombre es muy parecido al del **contexto de ejecución**, pero tenemos que saber que **son dos cosas distintas** y aprender a vivir con eso.

¿Qué eso de 'arguments' dentro del scope local de la función?

Es un objeto muy similar a los arrays que tienen **todas las funciones** en su scope local

Este objeto **contiene todos los parámetros** que recibe la función cuando fue **invocada**

¿Qué pasa si llamamos la misma función varias veces?



Como dijimos, cada vez que se invoca una función **se crea un contexto de ejecución** para esa función y se lo agrega a la pila y por más que se trate **de la misma función con el mismo nombre**, son dos elementos distintos en la pila de ejecución, cada uno **con su propio contexto de ejecución.**

Referencias

1. https://www.youtube.com/watch?v=yg8A5U7Wjsg8

2. https://medium.com/@happymishra66/execution-context-in-javascript-519dd72e8e2c

apuntes adicionales

Luis Andrés Villegas Sanchez

Contexto de ejecución

Se define como el entorno en el que se ejecuta el código de JavaScript. Por entorno, me refiero al valor de this, variables, objetos y funciones a las que el código JavaScript tiene acceso **en un momento particular**.

Si una variable u otra expresión no esta en el contexto de ejecución, entonces estas no están disponibles para su uso.

Contexto de ejecución global

este es el contexto de ejecución **predeterminado** en el que el código JS comienza su ejecución cuando el archivo se carga por primera vez en el navegador.

Todo el código global

Es el código(variables y funciones) que **no** está dentro de ninguna función u objeto, es decir, se ejecuta dentro del contexto de ejecución global

```
//global scope
var platzi = 'Nunca pares de aprender'

function saludar(nombre){
    // function scope
}
```

Contexto de ejecución global

no puede ser más de uno porque solo es posible un entorno global para la ejecución del código JavaScript ya que el motor JS tiene un solo subproceso

Contexto de ejecución funcional

El contexto de ejecución funcional se define como el contexto creado por el motor JavaScript cada vez que **encuentra alguna llamada a la función**.

Cada función tiene su propio contexto de ejecución

Puede ser más de uno

El contexto de ejecución funcional **tiene acceso a todo el código del contexto de ejecución global**, aunque viceversa no es aplicable

Referencias

https://medium.com/@happymishra66/execution-context-in-javascript-319dd72e8e2c

apuntes adicionales

5

contexto de ejecución

Scope

El scope, ámbito o alcance en JavaScript se refiere a **la accesibilidad o visibilidad de las variables**. Es decir, qué partes de un programa tienen acceso a la variable o dónde está visible la variable.

Puede ser una variable que hayamos definido

```
var platzi = 'Nunca pares de aprender'
```

Ó

el argumento de una función

```
function saludar(nombre){
    // function scope
}
```

Esto es muy importante porque **evita que reescribamos** el valor de una variable que ya habíamos definido

¿Por qué es importante el alcance?

1. El principal beneficio del alcance es la seguridad. Es decir, solo se puede acceder a las variables desde un área determinada del programa. Usando el alcance, podemos evitar modificaciones no deseadas a las variables de otras partes del programa.
2. El alcance también reduce las colisiones del espacio de nombres. Es decir, podemos usar los mismos nombres de variables en diferentes ámbitos.

```
//global scope
var nombre = 'Andres'

function saludar(){
    var nombre = 'Luis' //function scope
    console.log(nombre); //output Luis
}
saludar();
console.log(nombre)  //output Andres
```

Tipos de alcance

Hay tres tipos de alcance en JavaScript: 1) **Alcance global**, 2) **Alcance de función** y 3) **Alcance de bloque**, 4) **Module scope**

Alcance global

Windows es nuestro alcance global

Cualquier variable que no esté dentro de ninguna función o bloque (un par de llaves) está dentro del **alcance global**.

```
//Global scope
var message  = 'Hello, Platzi!'
```

Se puede acceder a las variables en el ámbito global desde **cualquier parte** del programa. Incluso corremos el riesgo de **modificar** alguna variable ya definida en este ámbito.

```
//Global scope
var message  = 'Hello, Platzi!'

//modificamos la variable global, 'S' de jQuery
var $ = function saludar(message) {
    //El scope de esta function, tiene
    //un parametro 'message' que es diferente a la variable
    //definida en el scope global
    console.log('Say: ', message)
}
```

alcance de la función

Las variables declaradas dentro de una función están dentro del alcance local (incluyendo los argumentos que se pasan a la función). **Solo se puede acceder desde esa función**, lo que significa que **no** se puede acceder desde el código externo.

```
function saludar(){
    var saludo = 'Hola'
    console.log(saludo)
}

//Print: Hola
saludar()

//Uncaught ReferenceError: saludo is not defined
console.log(saludo)
```

alcance de bloque

ES6 introdujo las variables **let** y **const**, a diferencia de las variables **var**, se pueden definir en el par de llaves más cercano. Eso significa que **no se puede acceder desde fuera de ese par de llaves**.

```
function printNumber(){
    for(let i = 0; i < 10; i++){ //loop for: es nuestro block scope
        //Cada vez que se ejecuta una iteración, se crea un block nuevo.
        //y en ese block, nos vamos a recordar del valor de 'i'
        setTimeout(() => {
            console.log(i)
        }, 100);
    }
}

printNumber();
```

Module Scope

Cuando se denota un script de tipo module con el atributo **type="module"** las variables **son limitadas** al archivo en el que están declaradas.

```
<script type="module" src="/assets/index.js"></script>
```

Ya no podemos acceder a las variables globales de este archivo desde la consola.

Ahora podemos separar código que esté en un archivo js usando **export** e **import**.

Alcance anidado

Al igual que las funciones en JavaScript, un ámbito se puede **anidar** dentro de otro ámbito

```
var name = 'Peter'                          Global scope

function greet(){                           function scope
    var greeting = 'Hello'
    {                                       block scope
        let lang = 'English'
        console.log(`${lang}: ${greeting} ${name}`)
    }
}

greet();
```

Aquí tenemos 3 ámbitos anidados uno dentro del otro. Primero, el alcance del bloque (creado debido a la variable let) está anidado dentro del alcance local ó de la función, que a su vez está anidado dentro del alcance global.

Referencias

https://blog.bitsrc.io/understanding-scope-and-scope-chain-in-javascript-f6637978cf53

NOTAS

Luis Andrés Villegas Sanchez

IIFE

Immediately-invoked function expressions,
es una función que se llama de la siguiente manera:

```
(function(){
    //function scope
    let color = 'green' //color ya no esta en un scope global

})();
```

La primera razón para usar una IIFE es por **la privacidad de los datos**.

¿Debo usar esto en mis proyectos actuales?

No. En ES2015 (inicialmente nombrado como ES6) es posible declarar ámbitos a nivel de bloque(let y const). En ES5 no, y es por ello que se emulaba este comportamiento con el uso de IIFEs.

apuntes adicionales

6

Closures

Junto a **scope** cuando lo combinamos con **function** podemos crear algo que se llama **closure**

Un closure **es una función interna** que tiene **acceso a el alcance de su función externa** incluso después de que la función externa ejecute un **return**.

```
function creaFunciones(){                    función externa
    var nombre = 'Luis'

    // 'muestraNombre' es una función interna (un closure)
    function muestraNombre(){      función interna
        console.log(nombre)
    }

    //Soy la referencia de una función 'muestraNombre'
    //y voy a recordar el estado de las variables
    //al momento que me invoquen :D
    return muestraNombre(también se puede hacer con objetos 😱)
}

var miNuevaFuncion = creaFunciones()
miNuevaFuncion(); //Print: Luis Mira, recordé el nombre 😎
```

¿Pero cómo el código de arriba funciona?

Una vez que **creaFunciones()** haya terminado de ejecutarse, es razonable suponer que **no** se pueda ya acceder a la variable nombre. Dado que el código funciona como se esperaba, esto obviamente no es el caso.

La solución a este rompecabezas es que miNuevaFuncion se ha **convertido** en un closure. Un closure es un tipo especial de objeto que combina dos cosas: **una función, y el ambito en que se creó esa función.**

Esto significa que un closure puede **recordar** y **acceder a variables** y **argumentos de su función externa** incluso después de que la función externa haya **finalizado**.

Recuerde

en JavaScript, el alcance de una variable se define por su **ubicación** dentro del código fuente y las funciones anidadas **(internas)** tienen acceso a las variables declaradas en su alcance **externo**.

Funciones que regresan funciones

Ya lo vimos en el ejemplo de arriba, pero vamos a mejorar algunas cosas:

```
function makeColorPrinter(color){
    let colorMessage = `The color is ${color}`;

    //voy a retornar la referencia de esta funcion
    //y voy a recordar la variable 'colorMessage' de
    //la funcion 'makeColorPrinter' y los argumentos
    //cuando me invoquen :D
    return function(){
        console.log(colorMessage)
    }
}

let greenColorPrinter = makeColorPrinter('green');
greenColorPrinter(); //Print: The color is green
```

Con los closures, podemos hacer variables 'privadas'

```
function makeCounter(n){
    let count = n  //Esta variable es privada.

    //los closures, ademas de ser funciones
    //tambien pueden ser un objeto.
    //Este objeto va a recordar todas las variables
    //definidas en el function scope
    return {
        increase: function(){
            count++
        },
        getCount: function(){
            return count
        }
    }
}

let counter = makeCounter(7)
console.log(counter.count) // Print: undefined. No esta en el scope global.
console.log('The count is: ', counter.getCount()) //Print: the count is: 7
```

Con este código, tenemos algo parecido a variables privadas, característica que no tiene JavaScript por default. Es decir, encapsulan variables que no pueden ser modificadas directamente por otros objetos, solo por las funciones pertenecientes al mismo

Resumen

Closure: Es **una función** o **un objeto con funciones** que recuerda el estado de las variables al momento de ser invocada, y conserva este estado a través de reiteradas ejecuciones.

Referencias

1. https://blog.bitsrc.io/a-beginners-guide-to-closures-in-javascript-97d372284dda

2. https://developer.mozilla.org/es/docs/Web/JavaScript/Closures

NOTAS

Luis Andrés Villegas Sanchez

This

En JavaScript siempre va a representar algo diferente dependiendo del **contexto** en el que se encuentre, ya sea en una función, objeto o clase.

'this' en el scope global

```
console.log(`this: ${this}`)
//Print: this: [object Window]
```

nos dirá que **this** es un **objeto Window**. Siempre que usemos **this** de forma global el browser lo va a **asignar** a Window.

'this' en el scope de una función

```
function whoIsThis(){
    return this
}

console.log(`whoIsThis: ${whoIsThis()}`)
//Print: whoIsThis: [object Window]
```

Al llamar la función directamente JavaScript le asigna Windows.

El motor de JavaScript le asigna este valor por **defecto**, excepto si estamos usando JavaScript en el modo **'use strict'**

'this' en el scope de una función en strict mode

```
function whoIsThisStrict(){
    'use strict'
    return this
}

console.log(`whoIsThisStrict: ${whoIsThis()}`)
//Print: whoIsThisStrict: undefined
```

Strict mode, nos sirve para evitar algunos errores

Errores que le puede pasar a cualquier programador

Básicamente, el motor de JavaScript intentará **ayudarnos** para evitar estos errores.

'this' en el contexto de un objeto

Se refiere a un objeto. Ese objeto es el que actualmente esta **ejecutando** un pedazo de código

Básicamente, **this** esta dentro de una función, pero esa función **pertenece a un objeto**, su valor cambia. En ese caso, **this** se refiere al objeto.

```
const person = {
    name: 'Luis',
    saludar: function(){
        console.log(`Hola soy ${this.name}`)
    }
}

person.saludar()
//Print: Hola soy Luis
```

'this' cuando sacamos la función saludar del objeto

```
const person = {
    name: 'Luis',
    saludar: function(){
        console.log(`Hola soy ${this.name}`)
    }
}

//pasamos la funcion como referencia
const accion = person.saludar
accion();
//Print: Hola soy
```

Esta función **acción**, **no** se esta llamando dentro del contexto de un objeto.

Es como **ejecutar** una función **directamente**.

this, en este caso, seria Windows

'this' en el contexto de una 'clase'

Las clases **no existen** en JavaScript

No son iguales a las clases de otros lenguajes de programación.

```
//Esta función funciona como un constructor
function Person(name){
    this.name = name
}

Person.prototype.saludar = function(){
    console.log(`Me llamo ${this.name}`)
}

const Luis = new Person('Luis');
Luis.saludar();
```

This, se refiere al objeto instanciado **'Luis'**

this.name = name lo hacemos sobre la instancia **Luis** no sobre el objeto prototipo

NOTAS

call, apply y bind

Estos métodos son parte del prototipo de function

Nos sirven para establecer el valor de this, es decir cambiar el contexto que se va a usar cuando la función sea llamada

call

Ejecuta una función, recibiendo como primer argumento el this(objeto) y los siguientes son los parámetros que recibe la función que llamo a call

```
function saludar(){
    console.log(`Hola, soy ${this.nombre} ${this.apellido}`)
}

const luis = {
    nombre: 'Luis',
    Apellido: 'Villegas'
}

//saludar.call(object, param1, param2, ... , paramN)
saludar.call(luis); //Print: Hola, soy Luis Villegas
```

Un ejemplo con argumentos

```
function caminar(metros, direccion){
    console.log(`${this.name} camina ${metros}m hacia ${direccion}`)
}

caminar.call(luis, 400, 'norte')
//Print: Luis camina 400m hacia norte
```

apply

Esta hace la misma funcionalidad de call, pero sus argumentos los pasamos de una forma ligeramente diferente.

Ejecuta una función, recibiendo como primer argumento el this(objeto) y el segundo parámetro un arreglo.

esto nos sirve cuando le tenemos que pasar una lista larga de parámetros a nuestra función

```
function caminar(metros, direccion){
    console.log(`${this.name} camina ${metros}m hacia ${direccion}`)
}

//caminar.apply(object, ...array)
caminar.apply(luis, [400, 'norte']) //Print: Luis camina 400m hacia norte
```

Trucos

Call → C de comas
esto por que le pasamos todos los valores separados por comas.

Apply → A de arreglo
acá le pasamos todos los parámetros por arreglos.

bind

A diferencia de los dos anteriores que vimos, bind no va a llamar a la función automáticamente.

```
const luis = {
    nombre: 'Luis',
    Apellido: 'Villegas'
}

function saludar(){
    console.log(`Hola, soy ${this.nombre}`)
}

saludar.bind(luis); //No imprime nada
```

.bind( ), me va a retornar una nueva función, con un nuevo contexto. Ese contexto se define en el primer parámetro.

```
function saludar(){
    console.log(`Hola, soy ${this.nombre}`)
}

// 'luisSaluda' es nuestra nueva funcion
const luisSaluda = saludar.bind(luis)
luisSaluda() // Print: Hola, soy Luis
```

Esta nueva función la ejecutamos colocando los paréntesis

También podemos pasar parámetros cuando llamamos con bind

```
function caminar(metros, direccion){
    console.log(`${this.name} camina ${metros}m hacia ${direccion}`)
}

const luisCamina = caminar.bind(luis, 400, 'norte')
luisCamina() //Print: Luis Camina 400m hacia norte
```

Function currying.

Esta técnica nos permite guardar argumentos parciales

```
function caminar(metros, direccion){
    console.log(`${this.name} camina ${metros}m hacia ${direccion}`)
}

const luisCamina = caminar.bind(luis, 400)
luisCamina('norte') //Print: Luis Camina 400m hacia norte
```

Nos dará el mismo resultado del código anterior, pero guardando ya un parámetro que es la distancia

Nos va a permitir hacer funciones reutilizables

NOTAS

Luis Andrés Villegas Sanchez

prototype

Un Prototipo **es un objeto** que existe como una propiedad **en cada función** en JavaScript.

```
function Human( ) {
  console.log(Human.prototype)
}
Human( )
```

1. Cuando se crea una función en JavaScript, el motor de JavaScript **agrega una propiedad prototype** a la función.

2. Si hacemos un console.log veremos que nos muestra **un objeto**

Console

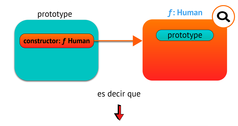
```
{constructor: ƒ}
```

Veamos más detallado esta propiedad del la función

El prototype objeto tiene una propiedad por defecto, llamada **constructor.**

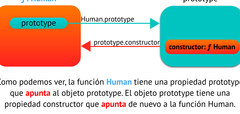
¿Qué es eso de constructor?

La propiedad **constructor es una función**



es decir que

La propiedad constructor, apunta de nuevo a la función Human



Como podemos ver, la función Human tiene una propiedad prototype que **apunta** al objeto prototype. El objeto prototype tiene una propiedad constructor que **apunta** de nuevo a la función Human.

¿Qué es eso de __proto__?

Los objetos **no tienen propiedad prototipo**. Tiene propiedad __proto__.

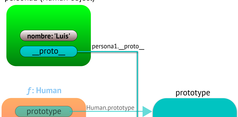
```
function Human(nombre) {
  this.nombre = nombre
}

const persona1 = new Humman('Luis');
```

1. Cuando se crea un objeto en JavaScript, el motor agrega una propiedad **__proto__** al objeto recién creado.

Console

```
console.log(persona1.__proto__);
{constructor: ƒ}
```

En este ejemplo __proto__ **apunta al objeto prototipo** de la función Human



La propiedad prototype de la función Human se comparte a **todos** los objetos **creados con la función constructora Human**

Objeto prototipo

Como prototipo **es un objeto**, podemos **adjuntar** propiedades y métodos al objeto prototipo.

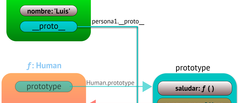
```
function Human(nombre) {
  this.nombre = nombre
}

Human.prototype.saludar = function ( ) {
  console.log('Hola, soy', this.nombre)
}

const persona1 = new Humman('Luis');
persona1.saludar();
```

1. Añadimos la propiedad saludar **al objeto prototipo.**

Console

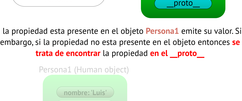
```
Hola, soy Luis
```

Veamos esto gráficamente

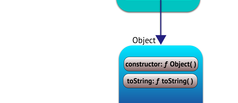


Persona1 **solo tiene** la propiedad nombre y __proto__. Entonces, ¿Cómo sale en consola el saludo?

Cuando intentamos acceder a una propiedad de un objeto, los motores de JavaScript primero intentan encontrar la propiedad en el objeto



Si la propiedad esta presente en el objeto Persona1 emite su valor. Sin embargo, si la propiedad no esta presente en el objeto entonces **se trata de encontrar** la propiedad **en el __proto__**



Si se encuentra la propiedad, se **devuelve su valor.** De lo contrario el motor de JavaScript nuevamente intentara **encontrar** la propiedad en el **siguiente** __proto__

Esta cadena de búsqueda continua hasta que la propiedad __proto__ sea **null**. En estos casos, se imprimirá en consola **undefined**

Referencias

1. https://medium.com/better-programming/prototypes-in-javascript-5bba2990e04b

apuntes adicionales

Luis Andrés Villegas Sanchez

prototype

En JavaScript todo son objetos, no tenemos clases, no tenemos ese plano para crear objetos.

Objeto común y corriente (Object literal)

Es una lista separada por comas de pares de nombre y valor envueltos en llaves.

Los objetos literales encapsulan datos, esto **minimiza** el uso de variables globales

```
const zelda = {
  name: 'Zelda'
}
zelda.saludar = function () {
  console.log( 'Hola soy ${this.name}' )
}
zelda.saludar()

const link = {
  name: 'Link'
}
link.saludar = function () {
  console.log( 'Hola soy ${this.name}' )
}
link.saludar()
```

1. De esta forma se crean objetos, se conoce con el nombre de **Object Literal**.
2. También podemos **agregar** atributos y funciones fuera del objeto.
3. Esto es algo ineficiente. Siempre que queramos añadir **nuevos objetos similares** nos tocara **volver** a escribir lo mismo

```
Console
Hola soy Zelda
Hola soy Link
```

Seamos un poco más eficiente que antes

Podemos crear una función que **nos construya** estos objetos que llamamos dos veces, así ahorrar tiempo y código.

```
function Hero(name) {
  const hero = {
    name: name
  }
  hero.saludar = function () {
    console.log( 'Hola soy ${this.name}' )
  }
  return hero
}

zelda = Hero( 'Zelda' );
zelda.saludar()

link = Hero( 'Link' );
link.saludar()

Console
Hola soy Zelda
Hola soy Link
```

1. Si la key y el value, **tienen el mismo nombre**, podemos colocar solo uno
2. La función saludar la estamos definiendo **cada vez** que llamamos la función Hero

Evitemos crear la misma función cada vez

Podemos crear un objeto aparte que tenga todas las funciones y que la función Hero **haga referencia** a ese nuevo objeto

```
const heroMethods = {
  saludar: function () {
    console.log( 'Hola soy ${this.name}' )
  }
}

function Hero(name) {
  const hero = {
    name: name
  }
  hero.saludar = heroMethods.saludar
  return hero
}

zelda = Hero( 'Zelda' );
zelda.saludar()

link = Hero( 'Link' );
link.saludar()

Console
Hola soy Zelda
Hola soy Link
```

1. Definimos un objeto con el método saludar
2. En lugar de definir la función saludar **cada vez**

```
hero.saludar = function () {
  console.log( 'Hola soy ${this.name}' )
}
```

3. Vamos a pasar la **referencia de la función saludar** del heroMethods. Por lo tanto, No estaremos creando a cada rato **la misma función** sino que haremos **referencia al mismo lugar de memoria**.

Object.create

Es uno de los métodos para crear un **nuevo objeto** en JavaScript.

El método acepta dos argumentos

```
Object.create(prototypeObject, propertiesObject)
```

prototypeObject

Si solo colocamos este parámetro, **me crea un nuevo objeto vacío**, pero con un prototipo definido

```
const heroMethods = {
  saludar: function () {
    console.log( 'Hola soy ${this.name}' )
  }
}

const hero = Object.create(heroMethods)
console.log(hero)
```

1. Si hacemos un console.log para ver que tiene el objeto

```
Console
▼ { }
  ▼__proto__: Object
    ▶ saludar: ƒ ()
    ▶ __proto__: Object
```

Esto viene del heroMethods

2. Como vemos, nos cree un **objeto vacío**, sin embargo, se **agrego** el contenido de heroMethods **en el atributo __proto__**

A esto se le llama herencia prototipal.

prototypeObject

Con este parámetro, podemos agregar atributos iniciales

Ahora mejoremos nuestro ejemplo que veníamos trabajando aplicando lo aprendido con **Object.create**

```
const heroMethods = {
  saludar: function () {
    console.log( 'Hola soy ${this.name}' )
  }
}

function Hero(name) {
  const hero = Object.create(heroMethods)
  hero.name = name
  return hero
}

zelda = Hero( 'Zelda' );
zelda.saludar()

link = Hero( 'Link' );
link.saludar()

Console
Hola soy Zelda
Hola soy Link
```

1. Creamos un objeto **vacío**, pero **con todas** las propiedades y métodos de heroMethods
2. Este objeto nuevo llamado **'hero'** le vamos a **agregar** un atributo name con su valor

```
▼ { name: 'link' }
  name: 'link'
  ▼__proto__: Object
    ▶ saludar: ƒ ()
    ▶ __proto__: Object
```

Prototype

Un Prototipo **es un objeto** que existe como una propiedad **en cada función** en JavaScript.

Como prototipo **es un objeto**, podemos **adjuntar** propiedades y métodos al objeto prototipo.

```
Object.create(prototypeObject, propertiesObject)

function Hero(name) {
  const hero = Object.create(Hero.prototype)
  hero.name = name
  return hero
}

Hero.prototype.saludar = function () {
  console.log( 'Hola soy ${this.name}' )
}

zelda = Hero( 'Zelda' );
zelda.saludar()

link = Hero( 'Link' );
link.saludar()

Console
Hola soy Zelda
Hola soy Link
```

1. Cuando se crea una función en JavaScript, el motor **agrega una propiedad prototype** a la función.
2. Utilizamos este objeto para añadir atributos y métodos

New

La keyword new **crea un nuevo objeto** que "hereda" todas las **propiedades** del prototype de otro objeto.

Todos los objetos "heredan" de un prototipo que a su vez hereda de otro prototipo y así sucesivamente creando lo que se llama la **prototype chain**

```
function Hero(name) {
  const hero = Object.create(Hero.prototype)
  this.name = name
  return this

Hero.prototype.saludar = function () {
  console.log( 'Hola soy ${this.name}' )
}

zelda = new Hero( 'Zelda' );
zelda.saludar()

link = new Hero( 'Link' );
link.saludar()

Console
Hola soy Zelda
Hola soy Link
```

1. Esta palabra **NEW** es azucar sintáctico, ya no tendremos que usar Object.create ni retornar el objeto Hero
2. Object.create ahora ocurre automáticamente No debemos indicarlo nosotros
3. Sirve imaginarnos que ya no será 'hero' sino **this**
4. El return this, es algo que ahora ocurre implícitamente, ya **no debemos colocarlo**

Al final podemos hacer lo mismo sin el keyword new pero usándolo nos facilitará muchas cosas. También existen otros keywords como class que nos ayudará mucho a ahorrar líneas de código.

NOTAS

Luis Andrés Villegas Sanchez

herencia prototipal

JavaScript no tiene clases como otros idiomas. Utiliza el concepto de prototipos y el encadenamiento de prototipos para la herencia.

Ya vimos como funciona la herencia, cuando creamos un nuevo objeto con la función Hero, pudimos acceder a el método saludar a pesar de que este no lo tiene.

```
function Hero(name) {
  this.name = name
}

Hero.prototype.saludar = function ( ) {
  console.log(`Hola soy ${this.name}`)
}

zelda = new Hero('Zelda');
zelda.saludar()
console.log(zelda)
```

en __proto__ se encuentra la función saludar

Console

```
Hola soy Zelda
▼ { name: 'zelda' }
  name: 'zelda'
  ▶ __proto__: Object
```

Cuando se llama a una función o variable que no se encuentra en el mismo objeto que la llamó, **se busca** en toda la **prototype chain** hasta encontrarla o regresar undefined

Propiedades de la instancia

```
zelda = new Hero('Zelda');
console.log("Name:", zelda.name)
```

Console

```
Name: Zelda
```

Propiedades de la 'clase'

```
zelda = new Hero('Zelda');
console.log("Saludar:", zelda.saludar)
```

Console

```
saludar: ƒ ( ) {
  console.log( `Hola soy ${this.name}`)
}
```

Propiedades heredadas (toString)

```
zelda = new Hero('Zelda');
console.log("toString:", zelda.toString)
```

Esta propiedad vienen desde Object

Console

```
toString: ƒ  toString ( ) { [native code] }
```

hasOwnProperty

Esto lo vamos a usar para saber si alguna propiedad **pertenece** a un objeto o **viene** de 'herencia'

```
console.log('hasOwnProperty("name"):', zelda.hasOwnProperty("name"))

console.log("hasOwnProperty("saludar"):", zelda.hasOwnProperty("name"))
```

La propiedad name si se encuentra en el objecto zelda

La propiedad saludar no se encuentra en el objecto zelda. Esta propiedad viene por herencia

Console

```
zelda.hasOwnProperty("name"): true
zelda.hasOwnProperty("saludar"): false
```

Analicemos a detalle todo lo relacionado al objeto 'zelda'

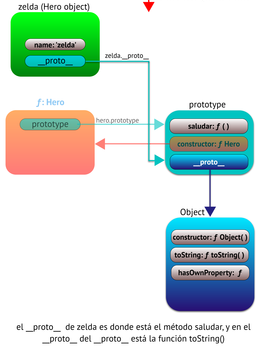
```
console.log(zelda)
```

Console

```
▼ Hero { name: "Zelda" }
  name: "Zelda"
  ▼ __proto__:
    ▶ saludar: ƒ  ( )
    ▶ constructor: ƒ  Hero(name)
    ▶ __proto__: Object
```

Lo primero que nos dice, es que viene de Hero, es la función que creo a zelda

Veamos esto gráficamente



el __proto__ de zelda es donde está el método saludar, y en el __proto__ del __proto__ está la función toString()

¿Qué hace el lenguaje para buscar una propiedad, por ejemplo saludar?

¿zelda.saludar existe?

No existe

Entonces ahora vamos a buscar en su __proto__

¿zelda.__proto__.saludar existe?

Si existe

Ok! entonces ejecutamos la función

Si se encuentra la propiedad, se **devuelve su valor**. De lo contrario el motor de JavaScript nuevamente intentara **encontrar** la propiedad en el **siguiente** __proto__

Esta cadena de búsqueda continua hasta que la propiedad __proto__ sea **null**. En estos casos, se imprimirá en consola **undefined**

La propiedad __proto__

Esta propiedad es algo que **puede** variar, no es algo que esta **específicado** en el lenguaje.

Es un detalle de implementación que ponen los browser, node o cualquier otro entorno

Object.getPrototypeOf

La forma correcta de acceder al proto es de la siguiente forma: object.getPrototypeOf

```
console.log(zelda.__proto__)
Object.getPrototypeOf(zelda)
```

```
{saludar: ƒ, constructor: ƒ }   console.log(zelda.__proto__)
{saludar: ƒ, constructor: ƒ }   Object.getPrototypeOf(zelda)
```

Vemos que nos imprime lo mismo en la consola usando cualquiera de estos dos métodos.

Object

Es el punto de partida de todos los objetos en JavaScript y eso incluye las funciones. Las funciones también son objetos en JavaScript

NOTAS

Luis Andrés Villegas Sanchez

¿Qué es el motor de JavaScript?

Es un programa o un **interprete** que ejecuta código JavaScript.

El motor es la parte encargada de **leer** y ejecutar el código fuente.

¿Qué es un interprete?

Es un software que **recibe** un programa en **lenguaje de alto nivel**, lo analiza y lo ejecuta.

> Los intérpretes informáticos, al contrario de los compiladores, no generan un fichero ejecutable, por lo que cada vez que se ejecuta un programa original debe pasar por la fase de análisis. Esto hace los intérpretes mas lentos que los compiladores. Wikipedia

Para analizar el programa completo, va traduciendo sentencias de código y ejecutándolas si están bien, así hasta completar el programa de origen.

Motores de JavaScript modernos (V8)

El Lanzamiento de V8 en 2008 marco un momento crucial en la historia de los motores. V8 reemplazo la relativamente **lenta** interpretación de JavaScript del navegador

La razón de esta mejora radica principalmente en la **combinación** de interprete y compilador: El interprete ejecuta el código fuente casi de **inmediato** y el compilador **traduce** el código fuente en código maquina que es sistema del usuario ejecuta de manera directa

A medida que el compilador genera el código de la maquina, aplica **optimizaciones**. Tanto la compilación como la optimización dan como resultado una ejecución de código mas rápida

La idea principal detrás de los motores modernos es combinar la mejor de ambos modelos:

- Rápido inicio del aplicación del interprete
- Rápida ejecución del compilador

Lograr ambos objetivos comienza con el interprete

El motor **marca** las partes del código ejecutadas con **frecuencia** como **"Hot Path"** (Ruta caliente) y las pasa al compilador junto con la información contextual recopilada durante la ejecución

Source code	JavaScript Engine
function getNameIperson(){ person.isAlive; }	Interpreter
const persons = [{ uno, {Daniel, ... }];	
for(let i = 0; i < 10000; i++){ getNameIperson(){}; }	Compiler (JIT)

Llamamos el **comportamiento** del compilador "just in time" o simplemente JIT

Referencias

https://www.campusmvp.es/recursos/post/fundamentos-de-javascript-por-que-deberias-saber-como-funciona-el-motor.aspx

Abstract Syntax Tree (AST)

Es una representación **jerárquica del programa** que presenta la estructura del código fuente de acuerdo con la gramática de un lenguaje de programación, cada nodo AST corresponde a un elemento de un código fuente.

Básicamente es un **desglose** basado en árbol de la estructura de su **código**, es decir los elementos en el código coinciden con los nodos en un árbol.

Veamos mas a fondo como funciona todo el proceso que hace el motor

El script se carga como un **flujo de bytes UTF-16**, ya sea para la red, la cache o un service worker, y se pasa a un decodificador de flujo de bytes

Ahora estos bytes se van a transformar en unos caracteres que nosotros establecimos en el código:

El valor decodificado será nuestro token

Básicamente, el decodificador de flujo de bytes decodifica los bytes en tokens. Los tokens se envían al analizador

Cuando el analizador ve que algo no tiene sentido, justo es cuando ocurre un **sintax error**

El analizador crea nodos en función de los tokens que recibe del decodificador de flujo de bytes. Con estos nodos, crea un Arbol de sintaxis abstracta o AST:

Es hora del interprete

El interprete **recorre** a través del AST y **genera el byte code** basado en la información que contiene el AST:

Aunque el código de bytes es rápido, puede ser **mas rápido**

A medida que se ejecuta este código de bytes, se genera **información**. Se puede **deducir** si cierto comportamiento ocurre con **frecuencia** y los tipos de datos que se han utilizado

Tal vez has estado invocando una función docenas de veces; ¿es hora de optimizar esto para que funcione aun más rápido?

El código de bytes, junto con la información generada se envía a un compilador de optimización

El byte code y el feedback type se **envian** al optimizing compiler, que genera código de máquina **altamente optimizado**

Ejemplo

Digamos que cierta función se invoca 100 veces y siempre he llamado con valores numéricos como argumentos cada vez

```
function sumi(x, y) {
  return x + y
}

sumi(1, 2)
```

¿Esto desvalive el número 3? La próxima vez que lo invoquemos, asumirá que lo estamos invocando nuevamente con dos valores numéricos.

Si eso es cierto, no se requiere una búsqueda dinámica, y solo puede **realizar** el código de máquina optimizado. De lo contrario, si la suposición era incorrecta, **volverá** al código de bytes original en lugar del código de máquina optimizado.

Por ejemplo, la próxima vez que lo invoquemos, pasamos una cadena en lugar de un número. Dado que JavaScript se escribe dinámicamente, podemos hacerlo sin ningún error.

```
function sumi(x, y) {
  return x + y
}

sumi('1', 2)
```

Esto significa que el número 2 se convertirá en una cadena, y la función devolverá la cadena en su lugar "2.2". Vuelve a ejecutar el bytecode interpretado y actualiza la retroalimentación de tipo.

Referencias

1. https://dev.to/fydalhalilee/javascript-visualized-the-javascript-engine-4cdf

2. https://v8.dev/

apuntes adicionales

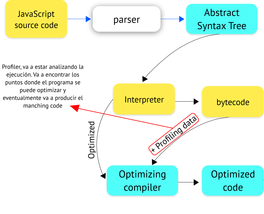
Jos Andrés Villegas Sanchez

Parsers y el Abstract Syntax tree

Para que el código llegue al navegador tiene que pasar muchas cosas, agarra el código, lo analiza, lo deconstruye, lo construye nuevamente, lo ejecuta y lo optimiza.

¿Qué hace un JavaScript Engine (motor)?

- Recibe código fuente
- El **parser** descompone en tokens, crea nodos en función de los tokens. Con estos nodos crea un Abstract Syntax Tree (AST)
- Se compila a bytecode y se ejecuta
- Se optimiza a machine code y se reemplaza el código base



¿Qué hace un parser?

Este agarra tu código fuente y **lo lee**, pero lo que tu escribes no es lo que la computadora entiende, primero lo tiene que descomponer, a estos pedazos se les llama **tokens**

Tokens, por ejemplo, **identifica que significa cada palabra clave** o símbolo. Una vez tenemos esos tokens se pasa al **Abstract Syntax Tree**.

SyntaxError

Que pasa si el parser esta **analizando** tu programa y de momento hay algo que no hace sentido, justo es cuando ocurre un **sintax error**

> Es lanzado cuando el motor de JavaScript se encuentra con partes de código que no forman parte de la sintaxis del lenguaje al momento de analizar el código. - MDN

No se puede leer tu código y no tiene sentido, los lenguajes de programación son muy formales, no se puede violar las reglas.

Google dice

- El **parser** es del 15% al 20% del proceso de ejecución
- La **mayoría** del JavaScript en una pagina nunca se ejecuta.
- Esto hace que **bundling y code splitting** sea muy importante!

bundling y code splitting

Esto significa que el código lo tenemos que empaquetar de una forma eficiente, es decir, separaremos el código y lo cargaremos cuando sea necesario. Esto se ve mucho en la **SPA**.

Parser de V8

Eager Parsing	Lazy Parsing
Encuentra errores de sintaxis	Doble de rápido que el eager parser
Crea el AST	No crea el AST
Construye scopes	Construye los scopes parcialmente

Eager Parsing

Cuando se hace este parsing vamos a encontrar todos los errores de sintaxis en el código que se esta analizando y vamos a **crear** el AST.

simplemente es una estructura en forma de árbol que representa tu programa, y además va ha **construir los scopes**.

En este momento vamos a saber que variables se pueden **leer** en que partes del código

Lazy Parsing

Cuando hacemos este parsing estamos **retrasando** alguna parte del código porque no hace falta analizarla aún

Esto tiene una ventaja y es que es el doble de rápido por lo tanto si las cosas tardaban 20% y logramos que ocurra mucho lazy parsing podemos retrasar ese análisis

Una consecuencia es que el AST **no se construye** y los Scopes se construyen **parcialmente**.

Veamos como se ven los Tokens

Abstract Syntax Tree (AST)

Esta es una estructura en forma de árbol, que **representa un programa**. Este empieza desde una raíz y se desploma en partes.

Se usa en:

- JavaScript Engine
- Bundlers: Webpack, Rollup, Parcel
- Transpilers: Babel
- Linters: ESlint, Prettify
- Type Checkers: TypeScript, Flow
- Syntax Highlighters

Se puede usar en muchos lugares, para corregir el código, para compilar, entre otras funciones. Podemos experimentar en el siguiente enlace.

Ejemplo

https://astexplorer.net/#/gist/47ea02626717418aeeffab28ccc47911/e3d8767b5e0436e3eda6c64e1745d5b1f83e3170

NOTAS

Luis Andrés Villegas Sanchez

Ejemplo

https://astexplorer.net/#/gist/7b068a69e1cdf45e246a8e9a42063546/06ebe63d07a2291b1e3bcc2aff2b10dece555360

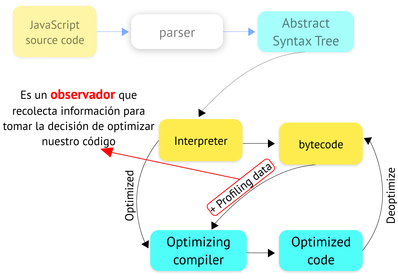
Abstract Syntax Tree

AST es una estructura que hace mucho mas fácil en análisis de código. Es la misma estructura que acabamos de usar para crear reglas de ESlint y la que el motor de JavaScript va a usar para leer nuestro código, convertirlo a bytecode y que pueda ser interpretado

NOTAS

Cómo funciona el JavaScript Engine

Justo después de que el Abstract Syntax tree(AST) se crea, se pasa el código a Bytecode.



Bytecode: es lo que va a estar ejecutando el motor de JavaScript y a su vez va a estar **produciendo** código optimizado.

Bytecode

Bytecode es parecido a assembly pero sobre un motor virtual

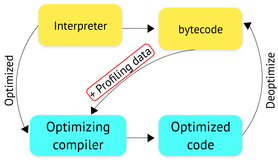
```
0x1117cfe2b55c @ 0 : 84 00 11 CreateFunctionContext [0], [15]
0x1117cfe2b561 @ 3 : 16 f6 PushContext r5
0x1117cfe2b563 @ 5 : 25 05 Ldar a0
0x1117cfe2b565 @ 7 : 1d 07 StaCurrentContextSlot [7]
```

Bytecode vs Machine code

Bytecode	Machine code
Código parecido a assemby	Binario
Portatil	Instrucciones especificadas a una arquitectura o procesador
Ejecutado por una virtual machine.	

profiling

Es un programa que esta en **medio** del bytecode y el optimizador



Lo que hace es que toma **observaciones** de la ejecución del bytecode y cuando ve que las funciones se están llamando igual, puede hacer algunas optimizaciones

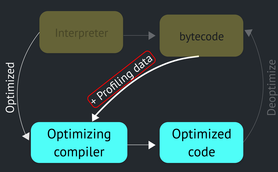
Ejemplo

```
function  add(a, b) {
    return a + b
}
for(let i = 0; i < 1000; i++){
    add(i, i)
}
```

1. Cuando esta función se llama muchas veces, se pondrá caliente
2. Cuando una **función está hot** significa que es hora de optimizar esa parte del código

```
function  add(a, b) {
    return a + b
}
```

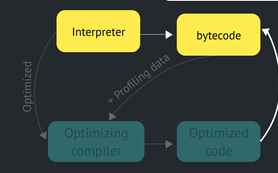
3. Allí es donde se pasa de bytecode a machine code



4. Si por casualidad ya no pasamos 2 números

```
add(1, 'uno')
```

5. Cuando la función recibe un parámetro diferente se quita la optimización y se pasa a bytecode



6. Básicamente se va a de optimizar el código

Por esto es bueno que las funciones se llamen **igual**, si le pasamos objetos que tengan una forma diferente o argumentos de tipos diferentes, entonces **no** se va a poder optimizar tu función o se va ha de optimizar

Los motores de JavaScript

cada browser tiene su propia implementación aunque todos siguen una estructura parecida

NOTAS

Luis Andrés Villegas Sanchez

event loop

El Event Loop hace que JavaScript parezca **multi hilo** a pesar de que corre un solo proceso.

JavaScript se ejecuta en el hilo principal del **navegador** de forma predeterminada

Básicamente, si un proceso tarda mucho **tiempo** en completarse toda la interfaz de usuario esta **atascada**.

El navegador

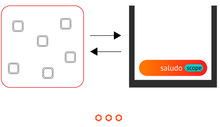
Nos brinda algunas **características** que el motor de JavaScript en si **no** proporciona: Son las Web APIs

Esto puede ayudarnos a crear un **comportamiento** asíncrono y sin bloqueo

Memory Heap

De forma desorganizada se guarda información de las variables, funciones y del scope

Al momento que ya tenemos, guardada una variable con su valor, una función y demás, entonces ya podríamos almacenar los registros al callStack



El scope

Esta determinado por el conjunto de variables, parámetros y funciones que se puede acceder desde esa **función** cuando se esta ejecutando.

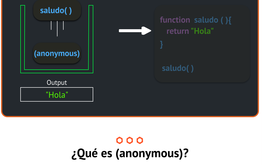
Stack

Va apilando de forma organizada las diferentes instrucciones que se llaman

Llevando así, un rastro de donde esta el programa, es decir en que punto de ejecución nos encontramos

El Stack es parte del motor de JavaScript, esto **no** es especifico del navegador

Cuando una función devuelve un valor (return) o termina de ejecutarse, se saca de la pila

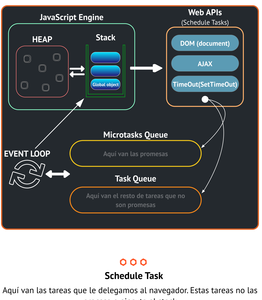


¿Qué es (anonymous)?

cuando ejecutamos un programa en JavaScript, la primara función que se agrega a la pila es una función **anónima** que engloba a todo el programa

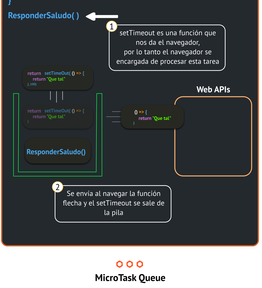
Significa que se **termino la ejecución** del programa principal

JavaScript Runtime



Schedule Task

Aquí van las tareas que le delegamos al navegador. Estas tareas no las procesa o ejecuta el stack



MicroTask Queue

Aquí se agregan las **promesas**. Esta Queue es la que tiene **mayor** prioridad.

Task Queue

Aquí se agregan las tareas que ya están listas para pasar al stack y ser ejecutadas. El stack debe estar vacío para que esto suceda

Event loop

Es un loop que se esta ejecutando todo el **tiempo** y pasa periódicamente revisando las queues y el stack

El Event Loop mira hacia el queue y al call stack. Si el callStack esta **vacio**, el event loop empuja el primer elemento del queue hacia el callStack

NOTAS

Luis Andrés Villegas Sanchez

Promesas

Pensamos las promesas como valores que **aun no** conocemos.

Las promesas se pueden resolverse de manera **síncrona y asíncrona**

Las promesas tienen **3 estados**

Estado Pending

Cuando creamos una promesa, la promesa esta en el estado pending

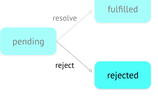
pending

Estado Fulfilled

Si la promesa se resuelve exitosamente, entonces va a pasar al estado de fulfilled

Estado rejected

Si ocurre algún error, en la acción asíncrona que esta resolviendo, entonces va a pasar al estado de rejected



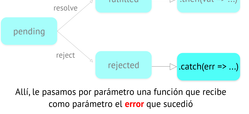
Para obtener de la resolución de la promesa

Podemos llamar a la función .then

Allí, le pasamos por parámetro una función que recibe como parámetro el **valor** que nosotros esperábamos

Si ocurre algún error

Si nunca se llega a resolver la promesa, podemos llamar la función .catch



Allí, le pasamos por parámetro una función que recibe como parámetro el **error** que sucedió

Promesas encadenadas

Luego de llegar al estado fulfilled, podemos retornar otra promesa **dentro del .then** y de esa manera ir encadándolas

Crear promesas

```
new  Promise(function(resolve, reject){

   if( todoOK ){
      resolve ( )  Se ejecuto el llamado exitosamente
   } else {
      reject ( )    Hubo un error en el llamado
   }

}).then( value => {
   Aquí ponemos nuestro código si se resolvió la promesa

}).catch( err => {
   Si ocurrió un error, es aquí donde lo trabajamos
})
```

async/await

Es solo **syntax sugar** de una promesa, por debajo es exactamente lo mismo

```
fetch(url)
  .then(response  => response.json())
  .then(data  => {
      console.log(data)

  })
```

```
const response = await fetch(url)
const data  = await response.json()
console.log(data)
```

VS

return en async/await

```
async function  consulta(url) {
   const response = await fetch(url)
   const data  = await response.json()
   console.log(data)
   const response = await fetch(url)
   return response.json()
}
```

1. Hacemos los mismo que en el ejemplo anterior

2. pero cuando hacemos un return, la palabra clave await queda implícita por lo tanto no es necesario ponerla

Promise.all

Permite ejecutar varias problemas en paralelo. Da error si alguna de las promesas es rechazada

```
Promises.all(ArrayDePromesas)
```

Cuando se resuelvan **todas** las promesas me regresa un arreglo con los valores que esperamos de cada una de las promesas. Promises.all es una **promesa** tambien

Promise.race

Regresa solo la promesa que se resuelva **primero**.

```
Promises.race(ArrayDePromesas)
```

array.slice

```
let numeros = [1, 2, 3]
let nuevoArray = numeros.slice(#-inicio, #-final)
```

devuelve una copia de una parte del array dentro de un nuevo array empezando por inicio hasta fin. El array original no se modificará.

for ... fo

```
for( let variable of arreglo){
      CODE ...
   }
```

La sentencia for...of (solo para arreglos) crea un bucle que itera a través de los **elementos de objetos iterables** (incluyendo Array, Map, Set, el objeto arguments, etc.), ejecutando las sentencias de cada iteración con el valor del elemento que corresponda.

Es posible usar const en lugar de let si no se va a **modificar** la variable dentro del bloque.

Referencias

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/for...of

NOTAS

Luis Andrés Villegas Sanchez

Getters y setters

Los getters y setters son funciones que podemos usar en un **objeto**. Para tener propiedades **virtuales**

se usan los keywords set y get para **crear** estas propiedades

```
const persona = {
   nombre: 'Luis',
   apellido: 'Villegas',
   get nombreCompleto ( ){
     return `${this.nombre} ${this.apellido}`
   },
   set nombreCompleto(value){

     const data = value.split(' ')
     this.name = data[0] || ''
     this.apellido = data[1] || ''
   }
}

console.log(persona.nombreCompleto)

persona.nombreCompleto = 'Daniela Sanchez'
console.log(persona.nombreCompleto)
```

1. Como persona es un objeto, **entonces** podemos usar uno de los features modernos que trae javascript que son getters y setters
2. Estas propiedades al ser **funciones** pueden llevar una validación de por medio
3. Se utilizan con el operador de asignación. Como **si fuera una variable más** dentro del objeto
4. De esta forma le pasamos un nuevo valor. Como si fuera una variable mas

Console

```
Luis Villegas
Daniela Sanchez
```

Después de la palabras clave get y set esta la propiedad con la que se relacionan (nombreCompleto)

Esta propiedad es una función que define **un comportamiento** cuando se accede a la propiedad o se modifica

Cuando se accede a la propiedad, se utiliza el valor de retorno del get. se usa para leer la propiedad; si no se especifica, el valor predeterminado de retorno es **undefined**

```
const persona = {
   nombre: 'Luis',
   apellido: 'Villegas',
   get nombreCompleto ( ){
     return `${this.nombre} ${this.apellido}`
   },
}

console.log(persona.nombreCompleto)
```

accedemos a la propiedad por lo tanto usamos el valor de retorno del get

Cuando se establece un valor, se llama al set y se le pasa el valor establecido

```
const persona = {
   nombre: 'Luis',
   apellido: 'Villegas',
   set nombreCompleto(value){

     const data = value.split(' ')
     this.name = data[0] || ''
     this.apellido = data[1] || ''
   }
}

persona.nombreCompleto = 'Daniela Sanchez'
```

Depende de nosotros lo que hagamos con ese valor. En este caso yo hice lo siguiente

Object.defineProperty

Para declarar getters y setters, también se puede **hacer** a través de Object.defineProperty

Object.defineProperty, define una nueva propiedad directamente en un objeto, o modifica una propiedad existente en un objeto, y devuelve el objeto

```
Object.defineProperty(objeto, nombrePropiedad, descriptor)
```

descriptor

Es un objeto que permite una adición o modificación precisa de una propiedad en un objeto.

```
const persona = {
   nombre: 'Luis',
   apellido: 'Villegas',
}

Object.defineProperty(persona, 'nombreCompleto', {

   get: function( ){
     return `${this.nombre} ${this.apellido}`
   },
   set: function( ){
     const data = value.split(' ')
     this.name = data[0] || ''
     this.apellido = data[1] || ''
   }
})
```

Cuando define una propiedad de esta manera, puede hacer **mucho más** que solo definir un setter o getter.

```
const persona = {
   nombre: 'Luis',
   apellido: 'Villegas',
}

Object.defineProperty(persona, 'edad', {

   enumerable:  false,
   configurable:  false,
   writable:  false,
   value:  42
})
```

enumerable

Si es verdadero, la propiedad aparecerá al **recorrer** el objeto con un loop, por ejemplo: for (var key in object). Por defecto es false.

Básicamente, la propiedad se muestra durante la enumeración de las propiedades

configurable

Si es true, el tipo de **descriptor** de propiedad puede **modificarse** y la propiedad puede ser **eliminada** del objeto. Por defecto es false

writable

Si es true, indica si el valor de la propiedad puede modificarse con el **operador de asignación**.

get

Una función cuyo valor retornado será el que se use como **valor** de la propiedad.

set

Una función que recibe como único argumento el nuevo valor que se desea asignar a la propiedad y que devuelve el valor que se almacenará finalmente en el objeto

Referencias

1. https://www.jackfranklin.co.uk/blog/es5-getters-setters/

2. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

NOTAS

Luis Andrés Villegas Sanchez

operador in

Devuelve true si la propiedad indicada esta en el objeto o en su prototipo

```
const persona = {
  nombre: 'Luis'
}
console.log('nombre' in persona )
```

propiedad in object

1. Se debe especificar un objeto en el lado **derecho** del operador in

Console

```
true
```

Propiedades heredadas

El operador in devuelve **true** para propiedades en la cadena del prototipo

```
'toString' in { }  // Devuelve true
```

Object.keys

El método Object.keys() devuelve un **array** de las propiedades **(key)** de un objeto, en el mismo orden como se obtienen en un loop normal.

```
const object = {
    key: value
}
```

Array.prototype.find

El método find() devuelve el **primer elemento** del arreglo que cumple con la condición proporcionada. En cualquier otro caso se devuelve **undefined**

```
var números = [1, 12, 4, 320, 30]
var elemento = números.find(numero => numero > 10)
console.log(elemento)
```

esta es la condición

Console

```
12
```

Referencias

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in

2.https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/find

3.https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/keys

apuntes adicionales

18

proxy

Igual que los getters y setters el proxy es uno de los features más recientes del lenguaje

El objeto Proxy se usa para definir un comportamiento **personalizado** para operaciones fundamentales

por ejemplo, para observar propiedades, cuando se asignan, invocación de funciones, etc

```
new Proxy(target, handler)
```

Básicamente, antes de que la llamada llegue al objeto original (target) podemos **manipularla** con una lógica que nosotros **definamos**

Target

Target es mi objeto a **supervisar.** Puede ser cualquier objeto, array, función o incluso otro proxy

Traps

Son los métodos **interceptores** que proveen acceso a las propiedades

Todas las trampas son opcionales. Si no se ha definido una trampa, el comportamiento predeterminado es **reenviar la operación** al **target**

Handler

Handler es un **objeto** con funciones (trampa) que definen el comportamiento del **proxy** cuando una **operación** es realizada en él.

```
var handler = {
  set(target, property, value){

  },
  get(target, property){

  }
}
var proxy = new Proxy(target, handler)
```

get y set son algunos de los métodos trampa del objeto handler

Target es el objeto original

property, puede ser o no **alguna propiedad** de nuestro objeto original

Cualquier operación que hagas en el proxy se enviara al objeto original

En este ejemplo, estamos utilizando un objeto JavaScript nativo al que nuestro proxy reenviará todas las operaciones que se le apliquen.

```
var target = {}
var proxy = new Proxy(target, { })
proxy.a = 37 // operación reenviada al objetivo
console.log(target.a)  // La operación ha sido reenviada correctamente
```

Podemos evitar este comportamiento con el handler

Console

```
37
```

Un handler, prácticamente recibe los cambios y administra esos cambios

Esto quiere decir, que si existe algo que tu no quieras en el **objeto original** (target) lo puedes descartar o si quieres mejorar algo antes que **afecte** al objeto original lo puedes mejorar también

Ejemplo

```
<script src="https://unpkg.com/fast-levenshtein@2.0.6/levenshtein.js"></script>

var target = {
  red: 'Rojo'
  green: 'green'
  blue: 'Azul'
}
var handler = {
  get(target, property){
    if(property in target){
      return target[property]
    }
    const suggestion = Object.keys(target).find( key =>
          Levenshtain.get(key, property) <= 3)
    if(suggestion){
      console.log( `${property} no se encontró. Quisiste decir ${suggestion}?`)
    }
    return target[property]
  }
}
var proxy = new Proxy(target, handler)
console.log(proxy)
console.log(proxy.red)
console.log(proxy.reeed)
```

1. levenshtain es un algoritmo que me va a permitir encontrar la distancia entre 2 cadenas
2. target será nuestro objeto original o el objeto base
3. handler también es un objeto y puede tener varias funciones (funciones trampa)
4. get recibe 2 cosas: el objeto target, que es nuestro **objeto original** y la propiedad que se esta **tratando de leer**
5. al objeto Proxy le pasamos el objeto original (target) y el objeto handler que va a definir el comportamiento del objeto Proxy

Console

```
▶Proxy { red: "Rojo", green: "Verde", blue: "Azul" }
Rojo
reeed no se encontró. Quisiste decir red?
```

Referencias

1.https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Proxy

2. https://www.youtube.com/watch?v=0o4-TvTkGtk

NOTAS

Luis Andrés Villegas Sanchez

Generadores

Los generadores son funciones especiales, pueden **pausar su ejecución** y luego **volver** al punto donde se **quedaron**

Su contexto (el valor de las variables, parámetros y funciones) será **conservado o recordado** al momento de retornar la ejecución de la función con **.next**

```
function*
```

1. Para crear un **generator function** , debemos agregarle un * pegado al **keyword function**

2. Ya luego escribimos el nombre de la función

```
function* generador( ){
  Aquí colocamos nuestro codigo
}
```

3. Ahora, el **objecto Generador** es retornado por una **generator function** y conforma tanto un protocolo iterable como un protocolo iterador

4. Al momento de **ejecutar** la función retorna un objeto, el **objeto generador**.

```
const gen = generador( )
console.log( gen )
```

Console

```
generador {<suspended>}
```

Características de los generadores

- Empiezan suspendidos y se tiene que llamar next para que se ejecute o continúe con la ejecución
- Regresan un value y un boolean que nos indica si ya se acabo o no la ejecución del generador.
- yield es la instrucción que **regresa** un valor cada vez que llamamos a **next** y detiene la ejecución del generador .

next

Retorna el valor ofrecido por la expresión **yield.**

Cada vez que llamamos next, la ejecución del generador va a continuar **hasta** el próximo yield

```
function* generador( ){
  console.log( 'GENERATOR START' )
    yield 1
    yield 2
    yield 3
  console.log( 'GENERATOR END' )
}

const gen = generador( )
console.log( gen.next() )
console.log( gen.next() )
console.log( gen.next() )
console.log( gen.next() )
```

Console

```
GENERATOR START
▶ {value: 1, done: false}
▶ {value: 2, done: false}
▶ {value: 3, done: false}
GENERATOR END
▶ {value: undefined, done: true}
```

Ejemplo - idMaker

```
function* idMaker( ){
  let id = 1
  while(true){
    yield id
    id++
  }
}

const gen = idMaker( )
console.log( gen.next() )
console.log( gen.next() )
console.log( gen.next() )
```

1. podemos hacer generadores infinitos

2. cuando generator function llegue al yield se va a detener, el while true no va a continuar

3. Se va detener y nos va a regresar el valor que este allí en el yield. Cuando le demos next va a continuar

4. Podemos seguir ejecutando next y nunca va a acabar. Esta en un while true y nuestro hilo principal no se bloquea

Console

```
▶ {value: 1, done: false}
▶ {value: 2, done: false}
▶ {value: 3, done: false}
```

Para reiniciar al ID debemos pasar un parámetro a .next

```
function* idMakerWithReset( ){
  let id = 1
  let reset
  while(true){
    reset = yield id
    if(reset){
      id = 1
    } else {
      id++
    }
  }
}
const gen = idMakerWithReset( )

console.log( gen.next() )
console.log( gen.next() )
console.log( gen.next() )
console.log( gen.next(true) )
```

1. el yield recibe **el valor** que pasamos por parámetro en el método .next

gen.next(true)

2. ese true va a establecer el valor de reset. Es como si **yield** fuese una función que regresa un valor

Console

```
{value: 1, done: false}
{value: 2, done: false}
{value: 3, done: false}
{value: 1, done: false}
```

Referencias

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Generador#Sintaxis

NOTAS

Luis Andrés Villegas Sanchez

API DEL DOM

Fetch

La API Fetch proporciona una **interfaz** para acceder y manipular partes del canal **HTTP**, como peticiones y respuestas.

Esta API provee un método global **fetch()** que proporciona una forma fácil y lógica de **obtener recursos** de forma asincrona por la red.

fetch historicamente proviene del XMLHttpReques (o XHR) con un funcionamiento similar al JQUERY.ajax

Una de las ventajas adicionales que se manejan con fetch vs XHR, es que fetch permite la cancelación de una señal (signal) mediante objetos de tipo AbortController.

El método fetch

Se implementa en múltiples interfaces, especificamente Window y WorkerGlobalScope. Esto hace que esté **método** se encuentre disponible en casi **cualquier contexto** donde quiera buscar recursos.

Básicamente fetch(), abrirá un canal de comunicación con un recurso **externo** (en red)

```
fetch(resource, init) : Promise<Response>

fetch(resource, init).then(function(response) {
        // Code
})
```

1. fetch, devuelve una **promesa**
2. Si se resuelve exitosamente la promesa, obtenemos el **objeto Response**
3. resource, es un argumento obligatorio. Puede ser una **URL** del recurso a solicitar o **un objeto Request**
4. init, es un objeto **opcional**
5. Se puede **acceder** a la respuesta o al objeto **Response** mediante un **then**
6. Una vez que Response es recuperada, hay **varios métodos disponibles**

Init

Este **objeto** contiene configuraciones para personalizar la solicitud. Estas opciones pueden ser:

- method: El método de solicitud, ejemplo: GET, POST, PUT, etc.
- headers: Cualquier cabecera que se quiera añadir a la solicitud, contenidas en un objeto Headers
- body: Cualquier cuerpo que se quiera añadir a la solicitud: puede ser un Blob, un objeto, etc. Las solicitudes con métodos GET o HEAD no pueden tener cuerpo
- signal: Una instancia del objeto AbortSignal; permite comunicarse con una solicitud vigente y **abortarla** si se desea vía AbortController.

```
var img = document.querySelector('img')
var cabecera = new Headers()
cabecera.append('Content-Type', 'image/jpeg')
var init = {
   method: 'GET',
   headers: cabecera,
   mode: 'cors',
   cache: 'default'
}

var request = new Request('flores.jpg')
fetch(request, init).then(function(response) {
   ...
})
```

1. En vez de colocar el objeto request también podemos **colocar una URL**

Cuando se rechaza una promesa

Se rechaza con un **TypeError** cuando sucede un error en la red, aunque normalmente significa un tema de permisos o similar.

El objeto Promise devuelto desde fetch() no será rechazado con un estado de error HTTP. Incluso si la respuesta es un error HTTP 404 o 500.

Este solo será rechazado ante un fallo de red o si algo impidió completar la solicitud.

Métodos del la API Fetch

Headers

Representa los encabezados de la respuesta/solicitud, lo que le permite consultar y tomar diferentes acciones en función de los resultados.

Request

Representa una solicitud de recursos.

Response

Representa la respuesta a una solicitud.

Response

Tú puedes crear un nuevo objeto Response

```
var response = new Response()
```

pero es más probable encontrar un objeto Response siendo devuelto como resultado de la operación de otra API, por ejemplo: fetch

Cómo has visto anteriormente, las instancias de Response son devueltas cuando fetch() es resuelto

Algunas propiedades de objeto response

- **Response.status** — Entero (por defecto con valor 200 si fue exitosa) que contiene el código de estado de las respuesta.
- **Response.statusText** — Cadena (con valor por defecto 'OK'), el cual corresponde al mensaje del estado de código HTTP.
- **Response.ok** — Visto en uso anteriormente, es una clave para comprobar que el estado está dentro del rango 200-299 incluido. Este devuelve un valor Boolean.

Algunas metodos de objeto response

Todas ellas devuelven una promesa que es eventualmente resuelta con el contenido actual.

- **Body.json( )** — Toma una secuencia Response y la lee hasta su finalización. Devuelve una promesa que si se resuelve exitosamente, obtenemos el objeto json
- **Body.blob( )** — Toma una secuencia Response y la lee hasta su finalización. Devuelve una promesa que si se resuelve exitosamente, obtenemos el objeto Blob
- **Body.text( )** — Toma una secuencia Response y la lee hasta su finalización. Devuelve una promesa que si se resuelve exitosamente, obtenemos un USVString(texto).
- **Body.formData( )** — Toma una secuencia Response y la lee hasta su finalización. Devuelve una promesa que si se resuelve exitosamente, obtenemos el objeto FormData

Referencias

1. https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Utilizando_Fetch

2. https://developer.mozilla.org/es/docs/Web/API/WindowOrWorkerGlobalScope/fetch

3. https://developer.mozilla.org/es/docs/Web/API/Fetch_API

4. https://developer.mozilla.org/en-US/docs/Web/API/Response

apuntes adicionales

Luis Andrés Villegas Sanchez

URL.createObjectURL

Este método estático crea un DOMString que contiene una URL que representa al objeto pasado como parámetro

```
var objectURL = URL.createObjectURL(object)
```

El parámetro object

Un objeto File o un objeto Blob para el que se **creara** la URL

DOMString

Es un String UTF-16. Dado que JavaScript ya usa estos string, se mapea DOMString **directamente a un String**

Pasarle null a un método o parámetro que acepte un DOMString suele convertirse a texto como "null"

Blob

Los Blobs representan datos que **no** necesariamente se encuentran en un formato nativo de JavaScript.

básicamente, blob es el binario de la petición que estamos pidiendo

```
const response = await fetch(url)
const blob = await response.blob()
```

Referencias

https://developer.mozilla.org/en-US/docs/Web/API/Body/blob

https://developer.mozilla.org/es/docs/Web/API/URL/createObjectURL

apuntes adicionales

20

API DEL DOM

AbortController.abort (es la forma de cancelar peticiones)

La interfaz AbortController representa un objeto que permite **cancelar** una o más solicitudes web cuando lo **desee**

```
var controller = new AbortController()
```

La **comunicación** con un request DOM se realiza mediante un objeto **AbortSignal**, antes de que se haya **completado**

Por ejemplo, podemos **comunicarnos** con un request fetch

Propiedad signal

Esta propiedad apunta al objeto **AbortSignal**. Este objeto se puede usar para **comunicarse** o **cancelar** un request DOM.

```
var controller = new AbortController()
controller.signal   // Solo lectura
console.log( controller.signal )
Console
▶AbortSignal {aborted: false, onabort: null}
```

Método .abort

Anula un **request DOM** antes de que se haya completado. Esto puede **abortar** los request de recuperación, el consumo de cualquier respuesta Body y las transmisiones.

```
var controller = new AbortController()
controller.signal
console.log( controller.signal )

controller.abort()
console.log( controller.signal )
Console
▶AbortSignal {aborted: false, onabort: null}
▶AbortSignal {aborted: true, onabort: null}
```

1 ahora ejecutamos el método **abort** para ver como se modifica la señal del **controller.signal**

Ejemplo - fetch

```
fetch(url, config)
// el segundo parámetro es opcional

const img = document.getElementById('huge-image')
var url = `https://images.pexels.com/photos/974470/
          nature-stars-milky-way-galaxy-974470.jpeg?q=100`
var controller
async function loadRequest( ){
  controller = new AbortController()
  try{
    const response = await fetch(url, { signal: controller.signal })
    const blob = await response.blob()

    const imgUrl  = URL.createObjectURL(blob)

    img.src = imgUrl
  } catch(err){
    console.log(err.message)
  }
}

function stopRequest( ){
  controller.abort()
}
```

1 Config es un objeto que contiene una **configuración personalizada** que desee aplicar a la **solicitud**

2 blob **es el binario** que estamos pidiendo

3 Con esto pasamos el binario a una URL que podemos utilizar después

4 Cuando abort se llama, la promesa fetch se **rechaza** con un AbortError

Cancelar peticiones

El hecho de poder cancelar peticiones con AbortController nos ayuda mucho en aplicaciones de una sola pagina (SPA)

Referencias

https://developer.mozilla.org/en-US/docs/Web/API/AbortController

NOTAS

Luis Andrés Villegas Sanchez

API DEL DOM

IntersectionObserver

Sirve para **observar** elementos, de forma asíncrona

Si este elemento que estamos observando cruza un umbral (la intersección del elemento con un elemento ancestro o con el viewport) que nosotros definamos nos lo va **notificar** para tomar alguna **acción**

Viewport

En términos de navegadores web, se refiere a la parte del documento que usted está **viendo**, la cual es actualmente **visible** en su ventana

El contenido fuera del viewport **no es visible** en la pantalla hasta que sea desplazado dentro de él.

Creando un Intersection Observer

Para crear una instancia o un nuevo objeto de este tipo llamamos a su constructor

```
var observer = new IntersectionObserver(callback, options)
```

pasándole por parámetro una callback para que se ejecute cuando el límite o el umbral (threshold) sea **cruzado** en una u otra dirección:

Opciones de Intersection Observer

El objeto options pasado al constructor IntersectionObserver()

```
var options = {
   root: document.querySelector('#element'),
   rootMargin: '0px',
   threshold: 1.0
}

var observer = new IntersectionObserver(callback, options)
```

1 Un threshold de 1.0 significa que cuando el 100% del elemento target está visible **dentro** del **elemento indicado** en root, el callback es invocado.

le deja **controlar** las circunstancias bajo las cuales el **callback** del observer es invocado.

¿Que son los atributos del objeto options?

root

Es usado como viewport para comprobar la **visibilidad** de elemento target.

Debe ser un elemento **ascendiente** (padre) del target. Por defecto se toma el viewport del **navegador si no se especifica** o si se especifica como null.

rootMargin

Puede tener valores similares a los márgenes en CSS. Este conjunto de valores sirve para aumentar o encoger cada lado del cuadro delimitador del elemento root antes de calcular las intersecciones.

Por defecto los márgenes son todos cero.

threshold

Es un **número** o un array de números que indican a que porcentaje de visibilidad **debe estar el elemento target** para que el callback del observer se **ejecute**

0 — 1

1 El porcentaje toma los números de [0, 1]

El valor por defecto es 0 lo que significa que tan pronto como **un pixel sea visible**, el callback del observer será ejecutado.

Un valor de 1.0 significa que el umbral no se considera pasado hasta que todos los pixels son visibles

Determinando un elemento para ser observado (target)

Una vez usted ha creado el observer, necesita darle un elemento target para observar

```
var options = {
   threshold: 0.25
}

var observer = new IntersectionObserver(callback, options)
var target = document.querySelector('#elemento')
observer.observe(target)   // El contenedor en este ejemplo es toda
                           // la pantalla del navegador

function callback(entries, observer){
   console.log( entries )
}
```

0 pasamos la referencia de la función, no la ejecutamos

1 Cuando el elemento **target** encuentra un **threshold** especificado por el IntersectionObserver, **la función callback es invocada**.

2 La función callback recibe una **lista**(arreglo) de objetos **IntersectionObserverEntry** y el observer

Console

```
▼[IntersectionObserverEntry]
  ▼0: IntersectionObserverEntry {time: ...
     time : <number>
     ▶rootBounds
     ▶boundingClientRect
     ▶intersectionRect
     isIntersecting : <bool>
     intersectionRatio : <number>
     ▶target: <#elemento>
     isVisible : <bool>
     ▶__proto__: IntersectionObserverEntry
     length: <number>
  ▶ __proto__: Array(0)
```

intersectionRatio : <number>

3 Nos dice cuanto porcentaje del elemento queda visible. Este valor lo vamos a utilizar para determinar si el elemento esta visible o no

intersectionRatio >= threshold decimos que esta visible el elemento

si especifica la opción root, el elemento target debe ser un descendiente(hijo) del elemento root.

Referencias

https://augdiaugus.gitbook.io/recoleccion-de-notas-publicas/escuela-de-javascript/curso-profesional-de-javascript#intersection-observer-api

NOTAS

Luis Andrés Villegas Sanchez

API DEL DOM

VisibilityChange

La API de Visibilidad de Página ofrece eventos que puedes escuchar para saber cuando un documento se convierte en visible o en oculto

Cuando el usuario minimiza la ventana o cambia a otra persaña, la API manda un evento de **visibilitychange**

```
document.addEventListener('visibilitychange', callback)
```

para que los listeners sepan que el estado de la página ha cambiado.

Objeto Document

El objeto document es la **página actual** que se está **visualizando** en ese momento.

```
console.log(document)
```

1 document Depende del objeto window, pero también puede depender del objeto frame en caso de que la página se está mostrando en un frame.

```
Console
▼#document
  <!doctype HTML>
  <html Lang = "en">
  ▶ <head>...</head>
  ▶ <body>...</body>
  </html>
```

Con el objeto document se **controla** la página web y **todos** los elementos que contiene.

Propiedades agregadas del objeto document

La API de visibilidad de página **agrega** las siguientes propiedades al Document:

Document.hidden

Devuelve true si la página se encuentra en un estado que se considera **oculto** para el usuario, y false lo contrario.

Document.visibilityState

indica el estado de visibilidad actual del documento, este estado se nos indica a través de una cadena de texto. Los valores posibles son:

```
document.visibilityState === 'visible'
```

El contenido de la página esta visible para el usuario. En la práctica, esto significa que la página es la pestaña de primer plano de una ventana no esta minimizada.

```
document.visibilityState === 'hidden'
```

El contenido de la página no es visible para el usuario, ya sea porque la pestaña del documento está en el fondo o parte de una ventana que está minimizada, o porque la pantalla del dispositivo está apagada.

```
document.visibilityState === 'prerender'
```

El contenido de la página se está procesando previamente y no es visible para el usuario.

Document.onvisibilitychange

Lo mencionamos al inicio del documento. Es Un EventListener que proporciona el código para llamar cuando el evento visibilitychange se dispara.

Ejemplo

```
document.addEventListener('visibilitychange', handleVisibilityChange)

function handleVisibilityChange(){
  const isVisible = document.visibilityState === 'visible'
  isVisible ? Video.play() : Video.pause()

}
```

1 Si el usuario esta **viendo** la pagina se ejecuta el video, **si no la esta mirando** entonces se pausa el video

Casos de uso

Consideremos algunos casos de uso para la API de Visibilidad de Página

- Un sitio tiene un carrusel de imágenes que no debería avanzar a la siguiente diapositiva a no ser que el usuario esté viendo la página.
- Una aplicación que muestra un panel de información y no se quiere que se actualice la información del servidor cuando la página no está visible.
- Una página quiere detectar cuando se está pre cargando para poder mantener un recuento preciso de las visitas de página.
- Un sitio desea desactivar los sonidos cuando el dispositivo está en modo de espera (el usuario presiona el botón de encendido para apagar la pantalla).

Referencias

https://developer.mozilla.org/es/docs/Web/API/Page_Visibility_API

NOTAS

Luis Andrés Villegas Sanchez

API DEL DOM

Service Workers

NOTAS

TypeScript

Introducción

TypeScript es un **lenguaje de programación** de código abierto desarrollado y mantenido por Microsoft.

Es un **superset** de JavaScript, añade **tipos** a nuestras variables y objetos basados en clases. Ayudando así a la detección de errores de forma temprana y mejorando el autocompletado.

En TypeScript, admitimos los mismos tipos de datos que esperaría en JavaScript, con un tipo de **enumeración** conveniente para ayudar a las cosas.

Qué es un Superset

Es un lenguaje escrito **encima** de otro lenguaje o mejor dicho, que **compila a otro lenguaje**. En el caso de TypeScript es un lenguaje que compila a JavaScript, pero que incluye muchas facilidades y ventajas.

Tenemos 2 forma de trabajar con TypeScript

La primera es instalando una herramienta llamada Parcel y la segunda es instalando TypeScript para luego compilarlo

Parcel

Para lograr trabajar con TypeScript necesitamos instalar un diferente tipo de servidor; live-server no es capaz de **transpirar** código typeScript a JavaScript. Para eso, vamos a **utilizar** una herramienta que se llama **Parcel**

> Transpilar es un término relativamente nuevo. Realmente es más correcto decir que TypeScript (u otros superset) transpilan, puesto que lo cierto es que no hay una compilación, sino es una traducción de un código fuente en otro código fuente en un lenguaje diferente.

Básicamente, los navegadores **no entienden TypeScript** asi que lo vamos a **transpilar** a JavaScript usando Parcel

Instalando Parcel

Para instalar Parcel utilizamos NPM

```
$ npm install -D parcel-bundler
```

Cuando Parcel termine de instalarse vamos a modificar algunas cosas en package.json

```
"scripts": {
  "start": "parcel file's.html"
},
"devDependencies": {
  "parcel-bundler": "^1.12.4"

},
"browserslist": [
  "last 1 Chrome version"
]
```

1. Añadimos esto a nuestro script para que inicie Parcel
2. Tenemos que especificar todos los archivos HTML que queramos que procésese Parcel
3. Con esta configuración hacemos que parcel solo transpile el código para el navegaro que estamos utilizando. browserslist es una indicación **a que Brower** queremos darle apoyo

Cuando iniciemos nuestro servidor

```
$ npm start
```

Parcel se encargara de construir nuestra aplicación y de configurar TypeScript solo

Cuando no cargue el servidor hacemos lo siguiente

1. Apagamos el servidor y luego borrar 2 carpetas

```
$ rm -rf .cache dist
```

2. .cache y dist

Instalando TypeScript via NPM

```
$ npm install -g typescript
```

Compilando nuestro código de TypeScript a JavaScript

Primero, creamos un **archivo.ts** y colocamos todo nuestro código de TypeScript

file.ts

```
console.log('Hello, TypeScript')
function add(a: number, b: number ){
  return a + b
}
const sum = add(2, 3)
```

1. en la línea de comandos ejecute el compilador de TypeScript

```
$ tsc file.ts
```

2. Esto nos genera un archivo JavaScript con todo el contenido de file.ts pero ya traducido a JavaScript

file.js

```
console.log('Hello, TypeScript')
function add(a, b){
  return a + b
}
const sum = add(2, 3)
```

¿Qué es el Análisis estático ?

Análisis estático de software es un tipo de análisis de software que se realiza **sin ejecutar el programa**

Referencias

1. https://desarrolloweb.com/articulos/introduccion-a-typescript.html

2. https://www.typescriptlang.org/docs/handbook/basic-types.html#introduction

3.https://parceljs.org/

NOTAS

Luis Andrés Villegas Sanchez

TypeScript

Tipos primitivos

Se llama tipo primitivo a los tipos de datos originales de un lenguaje de programación, esto es, aquellos que nos proporciona el lenguaje y con los que podemos (en ocasiones) construir tipos de datos abstractos y estructuras de datos.

Boolean

```
let muted = true
```

1. El boolean es el tipo de datos mas básicos. Sus posibles valores son: true o false.
2. Y con esto basta para TypeScript saber que muted es un booleano

```
let muted: boolean = true
muted = false
muted = 'Callado' ✗
```

3. Podemos ser **mas explícitos** y indicar que esta variable es de tipo booleano
4. Y si tratamos de asignar otro tipo de variable nos marca un error. Esto sucede porque TypeScript **hace un análisis estático de los tipos**

Numeros

```
let decimal: number = 6
let hex: number = 0xf00d
```

1. Todos los números en TypeScript son valores de coma flotante.
2. TypeScript también admite binarios y octales introducidos en ECMAScript 2015.

```
let binary: number = 0b1010
let octal: number = 0o744

let numero: number = 6 ✓
let numero: number = '6' ✗
```

3. Si ya le especificamos que la variable es de tipo numero no podemos asignar otro tipo de valor

String

```
let nombre: string = 'Luis'
let saludo = `Me llamo ${nombre}`
```

1. usamos el tipo string para referirnos a estos tipos de datos textuales.
2. TypeScript también utiliza comillas dobles (") o comillas simples (') para los string
3. Aunque la variable saludo no le especificamos el tipo, TypeScript sabe que es de tipo string

Arreglos

1. Los tipos de matriz se pueden escribir de una de dos maneras.

```
let people: string[] = ['Luis','Daniel','Brian']
```

2. En el primero, usa el tipo de elementos seguido de [] para denotar una matriz de ese tipo

```
people.push(900) ✗
```

3. Si ingresamos un numero tira un error ya que este arreglo solo acepta string y no números

```
let numeros: Array<number> = [1, 2, 3, 4]
```

4. Para tener un arreglo de string y de números, agregamos el operador (|)
5. La segunda forma usa un tipo de matriz genérico Array<elemType>

```
let peopleAndNumbers: Array<string | number> = []
peopleAndNumbers.push('Luis') ✓
peopleAndNumbers.push(900) ✓
```

Enums

```
enum Color {
  Rojo,
  Verde,
  Azul
}
```

1. Son un conjunto de valores
2. Una enumeración es una forma de dar nombres más amigables a conjuntos de valores numéricos.

```
let colorFavorito: Color = Color.Rojo
console.log(colorFavorito)
```

3. Por defecto, las enumeraciones comienzan a numerar a sus miembros a partir de 0

Console
```
0
```

4. También, puede cambiar esto configurando manualmente el valor de sus miembros.

```
enum Color {
  Rojo = 'Rojo',
  Verde = 'Verde',
  Azul = 'Azul'
}
```

5. Asignándole nuestros propios valores. Podemos colocar cualquier tipo de dato

```
let colorFavorito: Color = Color.Rojo
console.log(colorFavorito)
```

Console
```
Rojo
```

6. Una característica útil de las enumeraciones es que también puede pasar de un valor numérico al nombre de ese valor en la enumeración.

```
enum Color {
  Rojo,
  Verde,
  Azul
}
console.log(Color[1])
```

Console
```
Verde
```

Any

Cuando no tenemos la certeza de que tipo de variable será una variable, entonces podemos utilizar la anotación : any

Básicamente si queremos **inhabilitar el análisis estático de tipos** y dejar que los valores pasen por las comprobaciones, entonces etiquetamos a las variables con any

```
let comodin = "Joker"
comodin = { name: 'Luis' } ✗
```

1. Nos indica un error porque a la variable primero se le asigno un string, por lo tanto TypeScript entienda a la variable como un string y no como un objeto
2. Si nosotros anotamos a la variable comodin con **any**, entonces no habrá problema porque puede tomar cualquier valor

```
let comodin: any = "Joker"
comodin = { name: 'Luis' } ✓
comodin = 6 ✓
```

3. Estos valores pueden proveniir de contenido dinámico, por ejemplo, del usuario o de bibliotecas de terceros.

La anotación any es una forma poderosa de trabajar con JavaScript existente, lo que le permite activar y desactivar gradualmente la verificación de tipos durante la compilación.

Object

```
let person = { name: 'Luis' } ✓
let person: object = { name: 'Luis' } ✓
```

1. Esto funciona bien, pero podemos ser mas especificos y añadir la anotación : **object**, a nuestra variable
2. Los objetos son un tipo de dato no primitivo, es decir tipos de datos que no son originales del lenguaje. Es decir, todo lo que no es number, string, boolean, symbol, null, o undefined.

Tuple

Los tipos de tupla le permiten expresar una matriz con un número fijo de elementos cuyos tipos son conocidos

```
let x: [string, number]
let x = ["hello", 10]
console.log(x)
let x = ["Pepe", 20]
console.log(x)
```

1. Le asignamos los valores que espera la Tuple, en este caso el texto y el numero. Estos valores no necesitan ser los mismos

Console
```
▶ (2) ["Hola", 10]
▶ (2) ["Pepe", 20]
```

Referencias

https://www.typescriptlang.org/docs/handbook/basic-types.html#void

NOTAS

Luis Andrés Villegas Sanchez

TypeScript

Funciones

En TypeScript podemos ser explícitos con el tipo de los **argumentos** y el tipo de **retorno** de una función

Parámetros tipados

Podemos indicar a **cada** parámetro el **tipo de dato** que puede recibir

```
function suma(a, b){ ... }

function suma(a: number, b: number){
  return a + b
}

const sum = suma(2, 5)
const sum = suma(2, 'Pepe')

Error: 'Argument of type
"Pepe" is not assignable to
parameter of type number'

function suma(a: number, b: number): number
function suma(a: number, b: number): number{
  return a + b
}

function suma(a: number, b: number): number {
  return a + b
}
```

1. Escribamos una función que suma números.
2. Podemos trabajar así, pero TypeScript nos permite ser mas específicos indicando el tipo de variable
3. Si invocamos a esta función pasando por parámetro un valor **que no sea un numero**, entonces TypeScript nos avisara de un error
4. Si la función recibe 2 números y luego los suma y regresa su valor, TypeScript entiende que el valor retornado también **se trata de un numero**
5. Al momento de hacer **hover** a la función, muestra la función y nos indica **implícitamente** el valor de retorno, pero lo mejor sería especificar el tipo de retorno de la función
6. De esta forma indicamos el tipo de valor de retorno

Funciones que no retornan (void)

void es un poco como lo opuesto a any Por lo general, puede ver esto como el tipo de retorno de funciones que no devuelven un valor

```
function saludo() {
  console.log('Hola que tal')
}

function saludo(): void {
  console.log('Hola que tal')
}
```

1. Esta función no retorna ningún valor
2. Con TypeScript podemos ser mas explícitos indicando que no retorna nada con la anotación : void

Funciones que retornan un valor

```
function suma(a: number, b: number){
  return a + b
}

function suma(a: number, b: number): number {
  return a + b
}

function suma(a: number, b: number): number {
  return 'Hola mundo'
}

Type "Hola mundo" is not
assignable to type 'number'
```

1. Esta función nos retorna un numero
2. Podemos ser mas específicos y indicar el tipo de valor de retorno de esta función. En este ejemplo utilizaremos la anotación : number
3. Indicamos el tipo de retorno al final
4. Luego si la función retorna un tipo distinto a number se genera un error

Funciones que retornan un otra funciones

```
function createAdder(a: number) {
  return function(b: number){
    return b + a
  }
}

function createAdder(a: number): (b: number) => number {
  return function(b: number){
    return b + a
  }
}

const sum = fullName('Pepe', 2)
```

1. Podemos ser mas especificos y decirle a la función que tipo de valor es el que va a retornar
2. Como el valor de retorno es una función lo anotamos de esta forma : () => ...
3. Especificamos los parámetros y el tipo de valor de retorno

Parámetros opcionales y predeterminados

En TypeScript, cada parámetro es **requerido** por la función

Esto no significa que no se pueda dar null o undefined, sino que, cuando se llama a la función, el compilador **verificará** que el usuario haya proporcionado un valor para **cada** parámetro.

```
function fullName(firstName: string, lastName: string): string {
  return firstName + " " + lastName
}

const luis = fullName('Luis', 'Villegas')
const luis2 = fullName('Luis')

const sum = fullName('Pepe', 2)
```

1. El número de argumentos dados a una función tiene que **coincidir** con el número de parámetros que **la función espera,** de lo contrario TypeScript nos muestra un error
2. Recuerde que si invocamos a esta función y le pasamos por parámetro un valor que **no espera**, entonces TypeScript nos avisara de un error

En JavaScript, cada parámetro es opcional, y los usuarios pueden dejarlas como mejor les parezca. Cuando lo hacen, su valor es undefined

Podemos obtener esta funcionalidad en TypeScript agregando un ? al final de los parámetros que queremos que **sean opcionales.**

```
function fullName(firstName: string, lastName: string): string {
  return firstName + " " + lastName
}

function fullName(firstName: string, lastName?: string): string {
  if(!lastName){
    return firstName + " " + lastName
  } else {
    return firstName
  }
}

const luis = fullName('Luis', 'Villegas')
const luis2 = fullName('Luis')
console.log(luis)
console.log(luis2)

Console
Luis Villegas
Luis
```

1. Si queremos que el parámetro lastName sea opcional, entonces le agregamos el signo de interrogación ?

Cualquier parámetro opcional debe **seguir** los parámetros requeridos.

Si quisiéramos hacer que el primer nombre fuera opcional, en lugar del apellido, tendríamos que cambiar el orden de los parámetros en la función

Valores por defecto

Podemos establecer un valor que se **asignará** a un parámetro si el usuario **no proporciona** uno, o si el usuario pasa **undefined** en su lugar.

Estos se denominan parámetros por **defecto**.

```
function fullName(firstName: string, lastName: string) {
  return firstName + " " + lastName
}

function fullName(firstName: string, lastName: string = 'Smith') {
  return firstName + " " + lastName
}

const value = fullName('Luis', 'Villegas')
const value2 = fullName('Luis')
const value3 = fullName('Luis', undefined)

console.log(value)
console.log(value2)
console.log(value3)
Console
Luis Villegas
Luis Smith
Luis Smith
```

1. Tomemos este ejemplo y establecemos el apellido por defecto "Smith"

A diferencia de los parámetros opcionales, los parámetros por defecto no necesitan ocurrir después de los parámetros requeridos.

Si un parámetro por defecto viene antes que un parámetro requerido, los usuarios deben pasar explícitamente undefined para obtener el valor por defecto.

```
function fullName(firstName: string = 'Camilo', lastName: string) {
  return firstName + " " + lastName
}

const value = fullName('Luis', 'Villegas')
const value2 = fullName(undefined, 'Villegas')

console.log(value)
console.log(value2)
Console
Luis Villegas
Camilo Villegas

const value3 = fullName('Luis')   // error, muy pocos parámetros
```

Referencias

https://www.typescriptlang.org/docs/handbook/functions.html#optional-and-default-parameters

NOTAS

Luis Andrés Villegas Sanchez

TypeScript

Interfaces

Las interfaces nos permiten declarar la forma **exacta** de un objeto, no podemos añadir propiedades de mas, ni de menos. Además las interfaces definen los tipos de sus propiedades y si son opcionales o no

```
interfaces Rectángulo {
}
```

1. Para declarar una interfaz en TypeScript debemos usar la palabra clave Interfaces
2. Luego le damos un nombre a la interfaces y después colocamos las llaves { }

```
interfaces Rectángulo {
  ancho: number
  alto: number
}

let rect: Rectángulo
```

3. Dentro de la llaves, podemos declarar propiedades e indicar su tipo de variable
4. Cuando tenemos una interface se vuelve un tipo y podemos **anotar** las variables con este tipo de dato que acabamos de crear
   - **variable**: Rectangulo

○ ○ ○

Interfaces en objetos literales

```
interfaces Rectángulo {
  ancho: number
  alto: number
}

let rectángulo: Rectangulo = {
  ancho: 4
}

let rectángulo: Rectangulo = { ✗
  ancho: 4
}

Error: La propiedad 'alto' hace falta porque 'Rectangulo' la requiere

let rectángulo: Rectangulo = { ✓
  ancho: 4,
  alto: 6
}

let rectángulo: Rectangulo = { ✗
  alto: 6,
  ancho: 4
}

let rectángulo: Rectangulo = {
  alto: 6,
  ancho: 4,
  area: 24 ✗
}

Error: El objeto literal solo puede especificar propiedades conocidas, y 'area' no existe en el tipo 'Rectángulo'

let rectángulo: Rectangulo = {
  alto: 6,
  acho: 4 ✗
}

Error: El objeto literal solo puede especificar propiedades conocidas, y 'acho' no existe en el tipo 'Rectángulo'

let rectángulo: Rectangulo = {
  alto: 6,
  ancho: 4 ✓
}
```

1. Podemos crear nuestro objeto de esta forma y no tendremos ningún problema
2. Pero con TypeScript indicamos cual sería la estructura o que contenido obligatorio debe tener nuestro objeto, con las interfaces
3. Al hacer esto, nos marca un error ya que no tenemos dentro del objeto todas las propiedades de la interfaz
4. También podemos poner las propiedades en diferente orden y no pasa nada, simplemente debemos tener todas las propiedades de la interfaz
5. Si a un objeto lo anotamos con una interfaz este solo puede tener las propiedades definidas en la interfaz, **no puede tener mas** de las que se especificaron
6. Si colocamos una propiedad adicional que no esta en la interface, nos marca un error.
7. También ocurre un error sino escribimos los nombres de las propiedades del objeto exactamente igual como lo pusimos en la interfaz
8. Escribimos mal la propiedad 'ancho' y nos indica un error

○ ○ ○

Interfaces en parámetros de funciones

```
interfaces Rectángulo {
  ancho: number
  alto: number
}

function area(rect: Rectangulo) {
  return rect.alto * rect.ancho
}

let rectángulo: Rectangulo = { ✓
  ancho: 4,
  alto: 6
}

let área = area(rectángulo)
console.log(area)

Console
  24

let rectángulo = {
  ancho: 4,
  alto: 6,
  color: 'red'
}

function area(rect: Rectangulo) {
  return rect.alto * rect.ancho
}

let área = area(rectángulo)
console.log(area)

Console
  24

let rectángulo = {
  ancho: 4,
  color: 'red'
}

function area(rect: Rectangulo) {
  return rect.alto * rect.ancho
}

let área = area(rectángulo)

Falta la propiedad 'ancho' en el tipo '{alto: número; color: cadena; }' pero requerido en el tipo 'Rectangulo'.
```

1. Podemos indicar que el argumento **rect** solo reciba variables de tipo **Rectángulo**
2. Creamos un objeto y si lo anotamos con una interfaz(**Rectangulo**) este solo puede tener las propiedades definidas de esa interfaz, no puede tener mas ni menos de las que se especificaron
3. Aunque el parámetro sea de tipo Rectángulo, a TypeScript solo le interesa que el objeto tenga todos los elementos de la interfaz
4. A este objeto no lo anotamos con la interfaz : **Rectangulo** e incluso tiene mas propiedades
5. Aunque no especificamos que el objeto sea de tipo **Rectángulo** TypeScript no genera ningún error, porque el compilador solo verifica que el objeto tenga las propiedades requeridas y coincidan con los tipos requeridos
6. Si no le colocamos las propiedades requeridas nos genera un error

○ ○ ○

Propiedades opcionales

Las interfaces con propiedades opcionales se escriben de manera similar a otras interfaces, con cada propiedad opcional indicada por un **?** al final del nombre de la propiedad

```
enum Color {
  Rojo = 'Rojo',
  Verde = 'Verde'
}

interfaces Rectángulo {
  ancho: number
  alto: number
  color?: Color
}

let rectángulo: Rectángulo = { ✓
  ancho: 4,
  alto: 6,
  color: Color.Rojo
}

console.log(rectángulo.color)

Console
  Rojo

let rectángulo: Rectángulo = { ✓
  ancho: 4,
  alto: 6
}

console.log(rectángulo.color)
Console
  undefined
```

1. Podemos indicar que la propiedad color es opcional colocando un ? al final del nombre del atributo
2. Esto permitirá que el color sea totalmente opcional y podemos colocarlo o omitirlo
3. Si no colocamos el color no genera ningún error porque es un atributo **opcional**.
4. Si no tenemos el color entonces cuando lo imprimimos en consola nos saldrá **undefined**

NOTAS

Luis Andrés Villegas Sanchez

TypeScript

Clases

A partir de ECMAScript 2015 se añade el termino "clase" a JavaScript. TypeScript trabaja muy parecido a estas "clases" pero con ciertas ventajas, por ejemplo **encapsular**.

```
class Persona {
  private nombre: string
  private apellido: string
  constructor(nombre: string, apellido: string){
    this.nombre = nombre
    this.apellido = apellido
  }
  nombreCompleto(){
    console.log(`Mi nombre es: ${this.nombre} ${this.apellido}`)
  }
}

const luis: Persona = new Persona('luis', 'Villegas') ✔

function Persona(nombre, apellido){
  this.nombre = nombre
  this.apellido = apellido
}

Persona.prototype.nombreCompleto = function(){
  console.log(`Mi nombre es: ${this.nombre} ${this.apellido}`)
}

const luis: Persona = new Persona('luis', 'Villegas') ✘
```

1. Para crear una clase, escribimos la palabra reservada class y el nombre
2. Podemos tener variables privadas, publicas y protected
3. Si no se especifica el nivel de acceso en una variable o método, TypeScrit por defecto les pone public
3. Si **no** usamos el keyword class TypeScript **no** lo va a considerar como un tipo.
4. TypeScript nos indica un error porque Persona no es un tipo así sea que haga lo mismo que el código de arriba.

Herencia

```
class Animal {
  ...
}

class Dog extends Animal {
  ...
}
```

1. Para que una clase herede de otra usamos la palabra clave extends
2. Ahora la clase Dog puede acceder o usar a todos los atributos y métodos que tengan el nivel de acceso public o protected **de la clase** Animal

Modificadores publico (por defecto)

```
class Persona {
  nombre: string

  nombreCompleto(){
    console.log(`Mi nombre es: ${this.nombre} ${this.apellido}`)
  }
}

class Persona {
  public nombre: string

  public nombreCompleto(){
    console.log(`Mi nombre es: ${this.nombre} ${this.apellido}`)
  }
}
```

1. Si no especificamos los modificares en TypeScript, por **defecto**, cada miembro es public.
2. En nuestros ejemplos, hemos podido acceder libremente a los miembros que declaramos en todos nuestros programas. Esto se debe al modificador public
3. Si queremos, podemos ser mas especificos y añadir el modificar public al código anterior y tendremos exactamente el **mismo resultado**

Modificadores private

Cuando se marca un miembro private, no se puede acceder a ese miembro fuera de la clase, solamente desde ella misma

```
class Persona {
  private nombre: string
  constructor(nombre: string){
    this.nombre = nombre
  }
  nombreCompleto(){
    console.log(`Mi nombre es: ${this.nombre}`)
  }
}

const luis: Persona = new Persona('luis')

luis.nombre ✘
```

1. Dentro de la clase podemos utilizar la variable nombre sin ningún problema
2. Sin embargo fuera de ella no podemos y TypeScript nos muestra un error.

La propiedad 'nombre' es privada y solo es accesible dentro de la clase 'Persona'.

Modificadores protected

El modificador protected actúa de manera muy **similar** al modificador private con la excepción de que los miembros declarados como protected también pueden accederse dentro de las clases derivadas

```
class Persona {
  protected nombre: string
  constructor(nombre: string){
    this.nombre = nombre
  }
}

class Empleado extends Persona {
  private departamento: string
  constructor(nombre: string, departamento: string){
    super(nombre)
    this.departamento = departamento
  }

  saludo(){
    console.log(`Hola, nombre es: ${this.nombre}`)
  }
}
```

1. Podemos acceder a la variable en la clase empleado pero no fuera de la misma

NOTAS

Luis Andrés Villegas Sanchez

Patron de diseño

Los patrones de diseño son como **recetas** que resuelven problemas a los que nos afrentamos **frecuentemente** en el diseño de software

Los patrones de diseño son unas técnicas para **resolver** problemas **comunes** en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una **solución** a un problema de diseño.

Para que una solución sea considerada un patrón debe poseer ciertas características

Una de ellas es que debe haber comprobado su efectividad **resolviendo** problemas **similares** en ocasiones anteriores.

Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

En especifico es una solución para un problema dentro de un contexto

Este contexto **no** se vale que ocurra una sola vez, tiene que pasar **muchas veces**

Si ocurriera una sola vez, puede ser una solución, pero **no** lo podemos **categorizar** como un patrón diseño, los patrones de diseño van a resolver problemas dentro de un contexto recurrente.

En definitiva, una solución para un problema dentro de un contexto involucra lo siguiente:

- El contexto es la situación donde el **patrón** aplica.
- Esta situación debe ser **recurrente**
- Se refiere a la meta que estas tratando de alcanzar dentro del contexto.
- El problema incluye todas las **limitaciones** que existen dentro de este contexto
- La solución debe ser **genérica** que alcanza la meta dentro del contexto

Objetivos de los patrones

Los patrones de diseño pretenden:

- Proporcionar **catálogos** de elementos **reusables** en el diseño de sistemas software.
- Evitar la **reiteración** en la búsqueda de soluciones a problemas ya **conocidos** y solucionados anteriormente.
- Formalizar un **vocabulario** común entre diseñadores.
- **Estandarizar** el modo en que se realiza el diseño.
- Facilitar el **aprendizaje** de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.

Beneficios

Sirven como una **caja de herramientas** para soluciones bien **probadas** a problemas **comunes** en diseño de software

Proveen un lenguaje **común** que permite **comunicarse** de forma especifica y eficiente

Criticas

Los patrones de diseño introducen un nivel de complejidad. Por lo tanto, no vamos a estar constantemente buscando usar patrones de diseño, a no ser que sea **necesario**

> Los patrones, como todo tipo de complejidad, deben ser evitados siempre y cuando sea absolutamente necesario.
> - Jeff Atwwod

Lo **mejor** es tratar de solucionar los problemas de una forma **simple**, pero cuando ya vemos que **no hay una forma fácil,** entonces es allí cuando **podemos** ir a los patrones de diseño y ver si ya hay una solución que ya esta detallada que podemos aplicar.

Cuando estamos pensando todo el tiempo en patrones de diseño, es como dice el dicho "Cuando lo que tienes es un martillo, todo te parece un clavo". Solo debemos usarlo cuando sea necesario

Conclusión

- Siempre busca pensar en términos de diseño, no de patrones.
- Usa un patrón cuando hay una necesidad **natural** para usarlos
- Si existe una alternativa mas simple, prefiérela.

NOTAS

Luis Andrés Villegas Sanchez

Categorías de patrones de diseño

Patrones creacionales

Proveen diferentes mecanismos para crear objetos

Patrones estructurales

Describen formas de componer objetos para formas nuevas estructuras flexibles y eficientes

Patrones de comportamiento

Gestionan algoritmos y responsabilidades entre objetos

Patrones creacionales

NOTAS