Contenido

[Video 1 y 2 Introducción 1](#_Toc466221886)

[Video 3 BLOCK SCOPING WITH “LET” 2](#_Toc466221887)

[BLOCK SCOPING WITH “CONST” 3](#_Toc466221888)

[Video 4 Classes 4](#_Toc466221889)

[Video 5 Inheritance 5](#_Toc466221890)

[Video 6 Default funtion parameters 7](#_Toc466221891)

[Video 10 Modules 14](#_Toc466221892)

[Video 11 arrow Function or fat arrow 15](#_Toc466221893)

[Video 12 Introducción TypeScript 17](#_Toc466221894)

[Video 13 system part 1y 2 18](#_Toc466221895)

[Video 15. Funciones 19](#_Toc466221896)

[Video 16. Clases, objets, interfaces and inheritance 22](#_Toc466221897)

[Video 17 classes, objets interfaces and inheritace parte 2 25](#_Toc466221898)

[Video 18 modules 28](#_Toc466221899)

[Video 19. Decorators 31](#_Toc466221900)

Es un curso por el maestro Alejandro Rangel

# Video 1 y 2 Introducción

Son unos videos con cursos de Angular 2, TypeScript y ES6, contiene lecciones, teoría y ejercicios prácticos, pdf,ppt y código fuente., todo el trabajo basado a la arquitectura de Angular 2.

Se realizarán actividades como:

Weather App de búsqueda, y si introduces alguna información incorrecta marcara error.

Welcome to MyTunes: se harán búsquedas y se presentan las mas comunes, te dara enlaces que te llevaran a otra pantalla, veremos como realizar un login y password igualmente con verificación, si no estas registrado, aprenderemos a registrar el login.

Porque debemos tomar las clases:

Porque trabajar con Angular 2 es fácil y sencillo, el equipo estará desarrollando productos reales, y se trabajara con diferentes fuentes.

Compromisos del profesor Alejandro Rangel

Tendras toda la teoría, practica y tips. Las nuevas funciones de angular 2. Conferencias y asistencia personal.

Se estarán mirando: complex application, libraries, code generators y new features

Angular 2 es compatible con muchas aplicaciones (Edge, Firefox, Chrome, sefari, opera, etc) y se podrá reutilizar sus componentes las veces que lo necesites. Con soporte.

Cuando realizas una aplicación en angular 2, puedes verificar tu aplicación y evaluarla en ES6 Google Chrome canary.

# Video 3 BLOCK SCOPING WITH “LET”

Sintaxis

let var1 [= valor1] [, var2 [= valor2]] [, ..., varN [= valorN]];

Parámetros

var1, var2, …, varN

Nombre de la variable. Puede ser cualquier identificador de variable legal.

value1, value2, …, valueN

Valor inicial de la variable. Puede ser cualquier expresión legal.

Descripción

let permite declarar variables limitando su alcance (scope) al bloque, declaración, o expresión donde se está usando. Lo anterior diferencia  la expresión let de la palabra reservada [var](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/var) , la cual define una variable global o local en una función sin importar el ámbito del bloque.

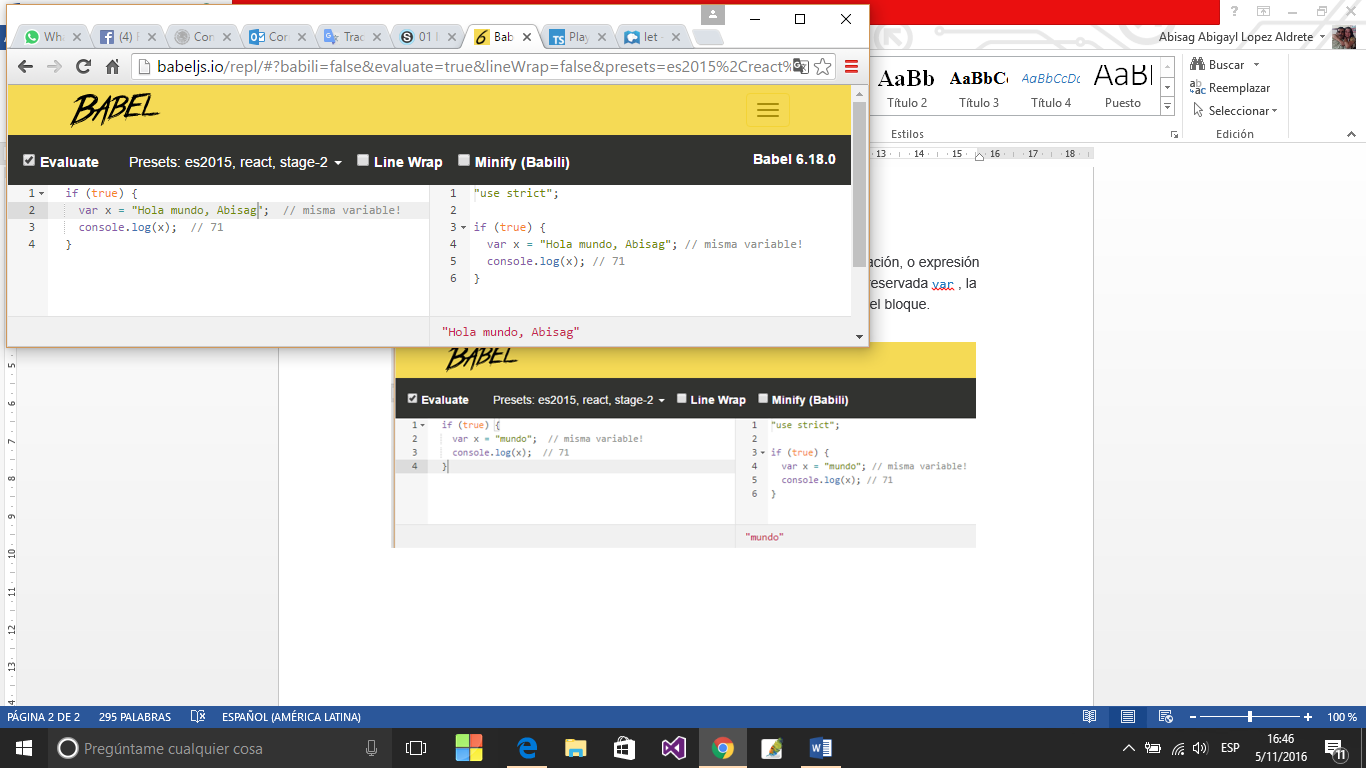
Ejemplo:

if (true) {

var x = "Hola mundo, Abisag"; // misma variable!

console.log(x); // 71

}



# BLOCK SCOPING WITH “CONST”

Las constantes son bloques de ámbito, al igual que las variables definidas mediante el [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) comunicado. El valor de una constante no puede cambiar a través de re-asignación, y no se puede redeclarada.

Sintaxis

const nombre1 = valor1 [ , nombre2 = valor2 [ , ... [ , nombre- = valorN]]] ;

nameN

El nombre de la constante, que puede ser cualquier legal identificador.

valueN

El valor de la constante; esto puede ser cualquier legal expresión.

Descripción

Esta declaración crea una constante que puede ser global o local a la función en la que se declara. Se requiere un inicializador para una constante; es decir, debe especificar su valor en el mismo comunicado en el que se declaró (lo cual tiene sentido, dado que no se puede cambiar más adelante).

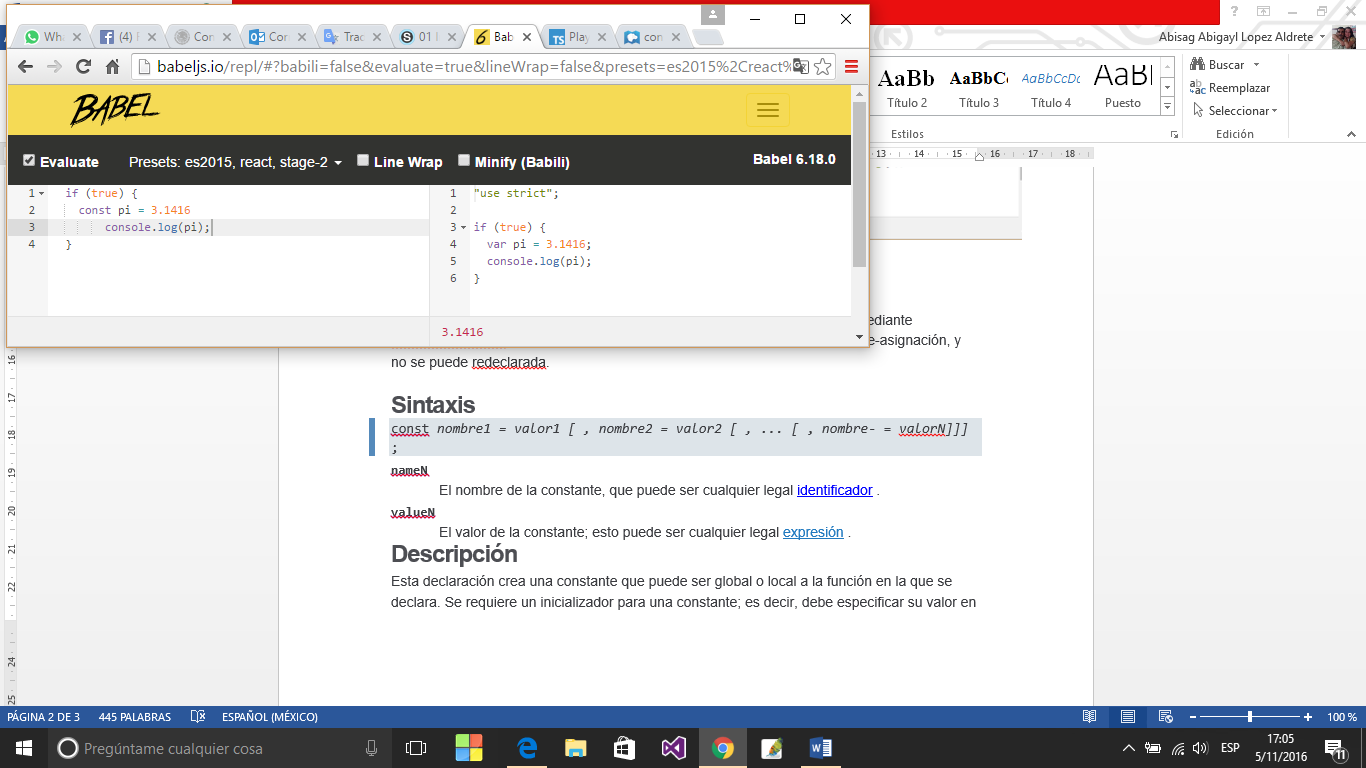
Ejemplo:

if (true) {

const pi = 3.1416

console.log(pi);

}



# Video 4 Classes

La declaración de classes crea una nueva clase con un nombre dado utilizando la herencia basada en prototipo.

Sintaxis

class name [extends] {

// class body

}

Descripción

Al igual que con las expresiones de class, el cuerpo de class de una declaración de clase se ejecuta en modo estricto. La propiedad constructor es opcional.

Ejemplo:

class person {

constructor(name,founder, year) {

this.name = "Abisag";

this.founder = founder;

this.year = year;

}

summary(){

return this.name + "fue creado por" + this.founder + this.year;

}

}

Ejemplo 2

class person {

name;

constructor(name) {

this.name = name;

}

getname(){

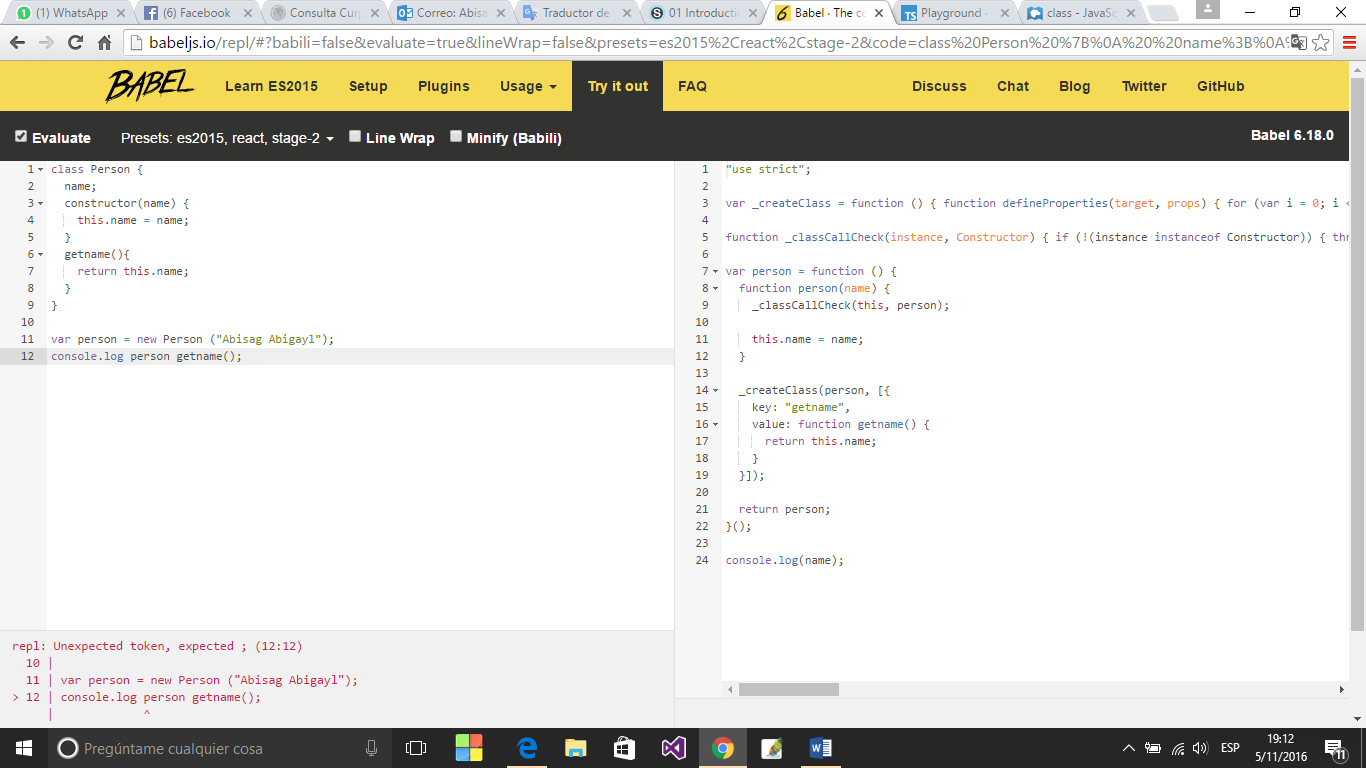
return this.name;

}

}

var person = new person ("Abisag Abigayl");

console.log person getname();



# Video 5 Inheritance

classical inheritance

Herencia de prototipos de JavaScript tiene un poder más expresivo que la classical inheritance, como veremos en la actualidad.

Pero en primer lugar, ¿por qué nos importa la herencia en absoluto? Hay principalmente dos razones. La primera es escribir conveniencia. Queremos que el sistema de la lengua de forma automática fundidoreferencias de clases similares. Poco tipo de seguridad se obtiene a partir de un sistema de tipo que requiere la conversión explícita rutina de referencias a objetos. Esto es de importancia crítica en lenguajes de tipo fuerte, pero es irrelevante en lenguajes de tipo débilmente como JavaScript, donde las referencias a objetos nunca necesitan de fundición.

La segunda razón es la reutilización de código. Es muy común tener una cantidad de objetos de aplicación todo exactamente los mismos métodos. Las clases hacen que sea posible crear todos ellos desde un único conjunto de definiciones. También es común tener objetos que son similares a algunos otros objetos, pero que sólo difieren en la adición o modificación de un pequeño número de métodos. herencia clásica es útil para esto, pero herencia de prototipos es aún más útil.

Ejemplo:

Function.method('inherits', function (parent) {

this.prototype = new parent();

var d = {},

p = this.prototype;

this.prototype.constructor = parent;

this.method('uber', function uber(name) {

if (!(name in d)) {

d[name] = 0;

}

var f, r, t = d[name], v = parent.prototype;

if (t) {

while (t) {

v = v.constructor.prototype;

t -= 1;

}

f = v[name];

} else {

f = p[name];

if (f == this[name]) {

f = v[name];

}

}

d[name] += 1;

r = f.apply(this, Array.prototype.slice.apply(arguments, [1]));

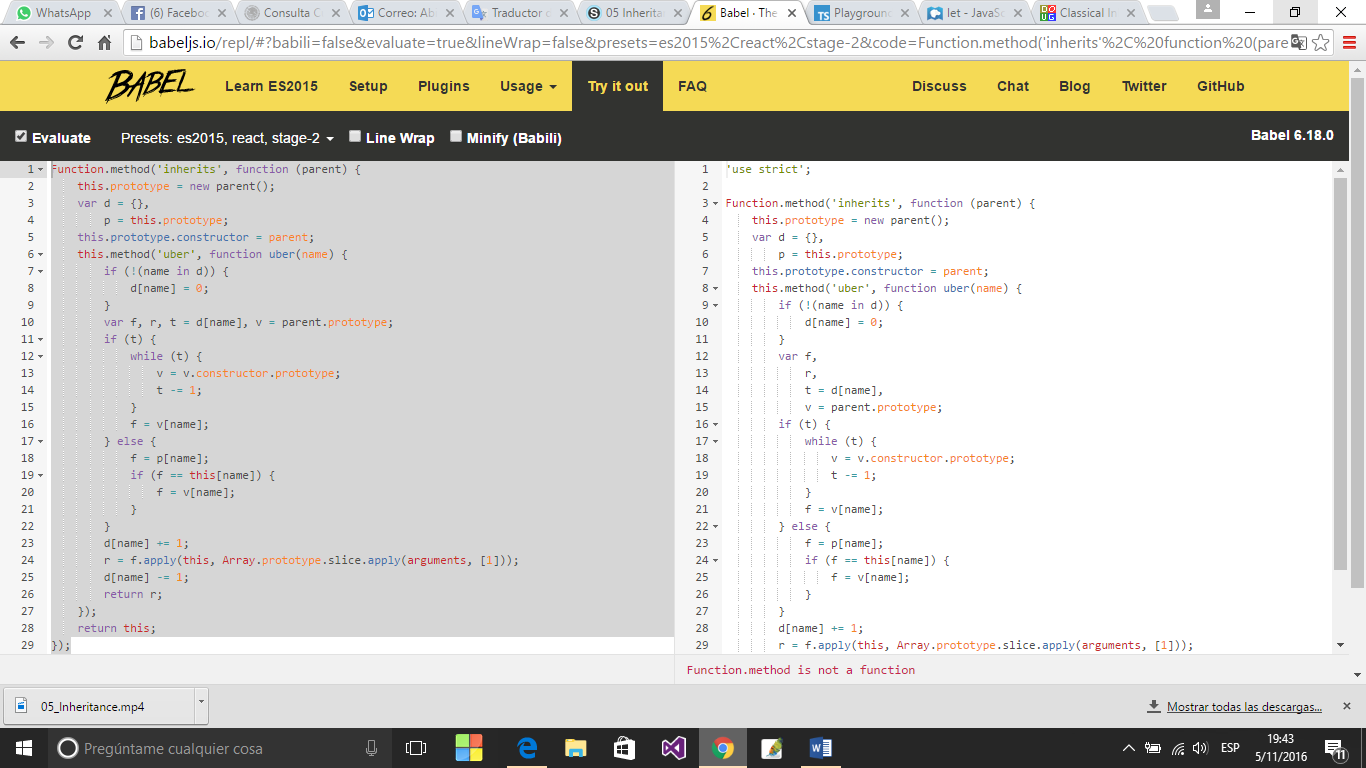
d[name] -= 1;

return r;

});

return this;

});



# Video 6 Default funtion parameters

Los parámetros por defecto de una función permiten que los parámetros formales de la función sean inicializados con valores por defecto si no se pasan valores o los valores pasados son undefined.

Sintaxis

function [nombre]([param1[ = valorPorDefecto1 ][, ..., paramN[ = valorPorDefectoN ]]]) {

declaraciones

}

Descripción

En JavaScript, los parámetros de funciones son por defecto undefined. De todos modos, en algunas situaciones puede ser útil colocar un valor por defecto diferente. Aquí es donde los parámetros por defecto pueden ayudar.

En el pasado, la estrategia general para colocar valores por defecto era probar los valores de los parámetros en el cuerpo de la función y asignarles un valor si eran undefined. Si en el siguiente ejemplo no se proveyera un valor para b en la llamada, su valor sería undefined cuando se evalúe a\*b y la llamada a multiplicar hubiera retornado NaN. De todos modos, esto es capturado en la segunda línea de este ejemplo:

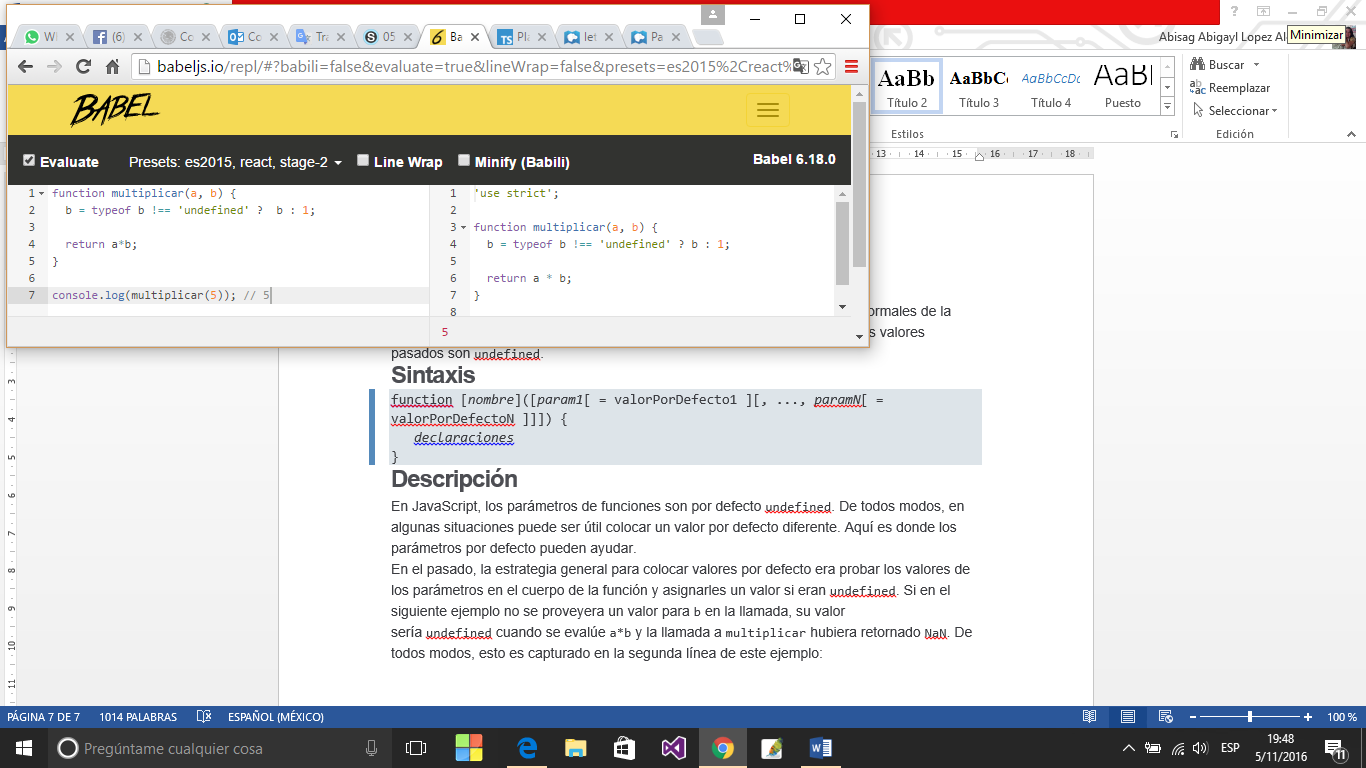
function multiplicar(a, b) {

b = typeof b !== 'undefined' ? b : 1;

return a\*b;

}

console.log(multiplicar(5)); // 5



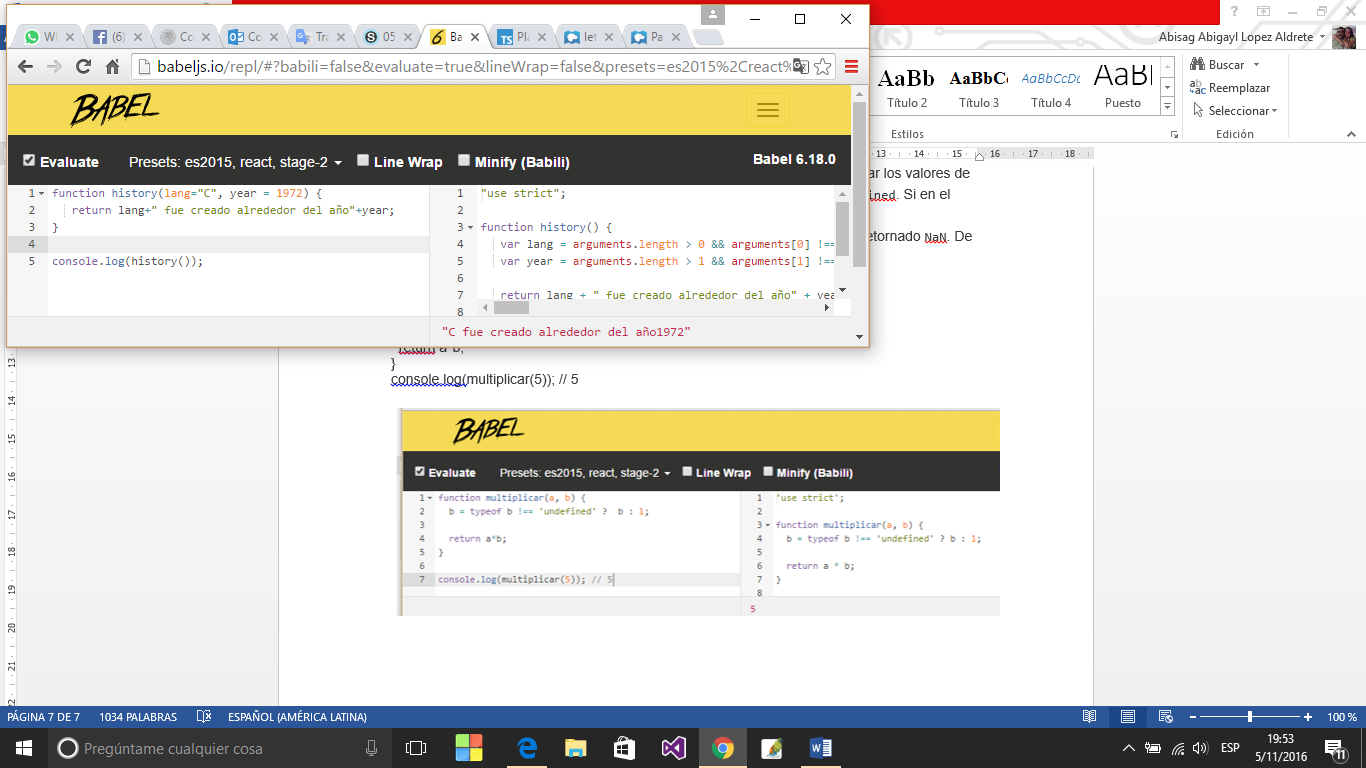
Pasando parámetros:

function history(lang="C", year = 1972) {

return lang+" fue creado alrededor del año"+year;

}

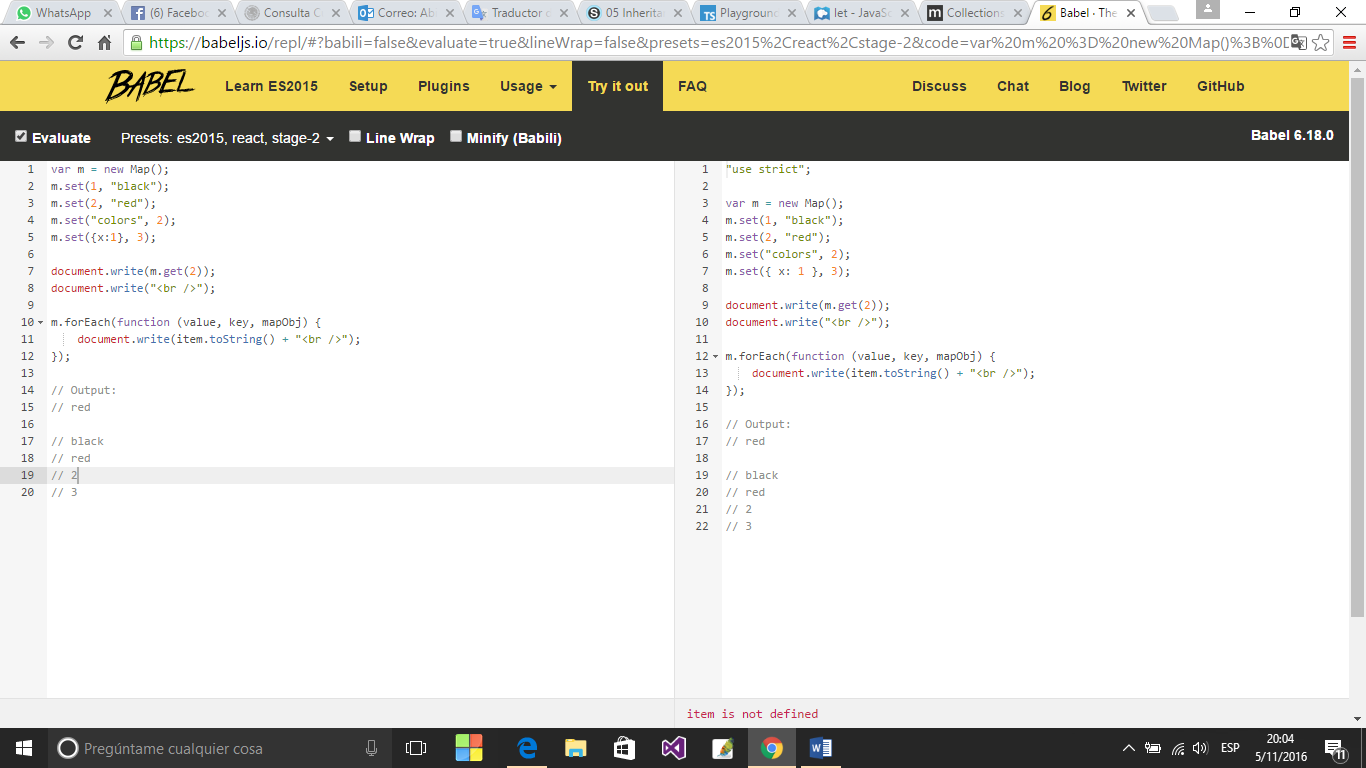
console.log(history());



Video 7 Collections (JavaScript)

El map y WeakMap objetos pares clave / valor de la tienda y le permiten añadir, recuperar y eliminar miembros mediante el uso de la clave. La clave y el valor pueden ser de cualquier tipo. Los Set objeto almacena valores de cualquier tipo.

Los Map y Set objetos le permiten enumerar miembros de la colección mediante el uso de la forEach método y comprobar el tamaño de la colección utilizando el tamaño de método. El WeakMap objeto, por el contrario, no es enumerable. Para esta colección, las referencias clave se mantienen débilmente. Utilice WeakMap si desea que el recolector de basura para determinar si la aplicación tiene que retener a cada miembro de la colección en la memoria. Por ejemplo, esto puede ser útil en escenarios de almacenamiento en caché donde los objetos almacenados en caché son muy grandes y que no quieren sujetar objetos en la memoria innecesariamente. En algunos casos, puede utilizar este objeto para evitar pérdidas de memoria.



var engines = new Set(); // crea un nuevo set

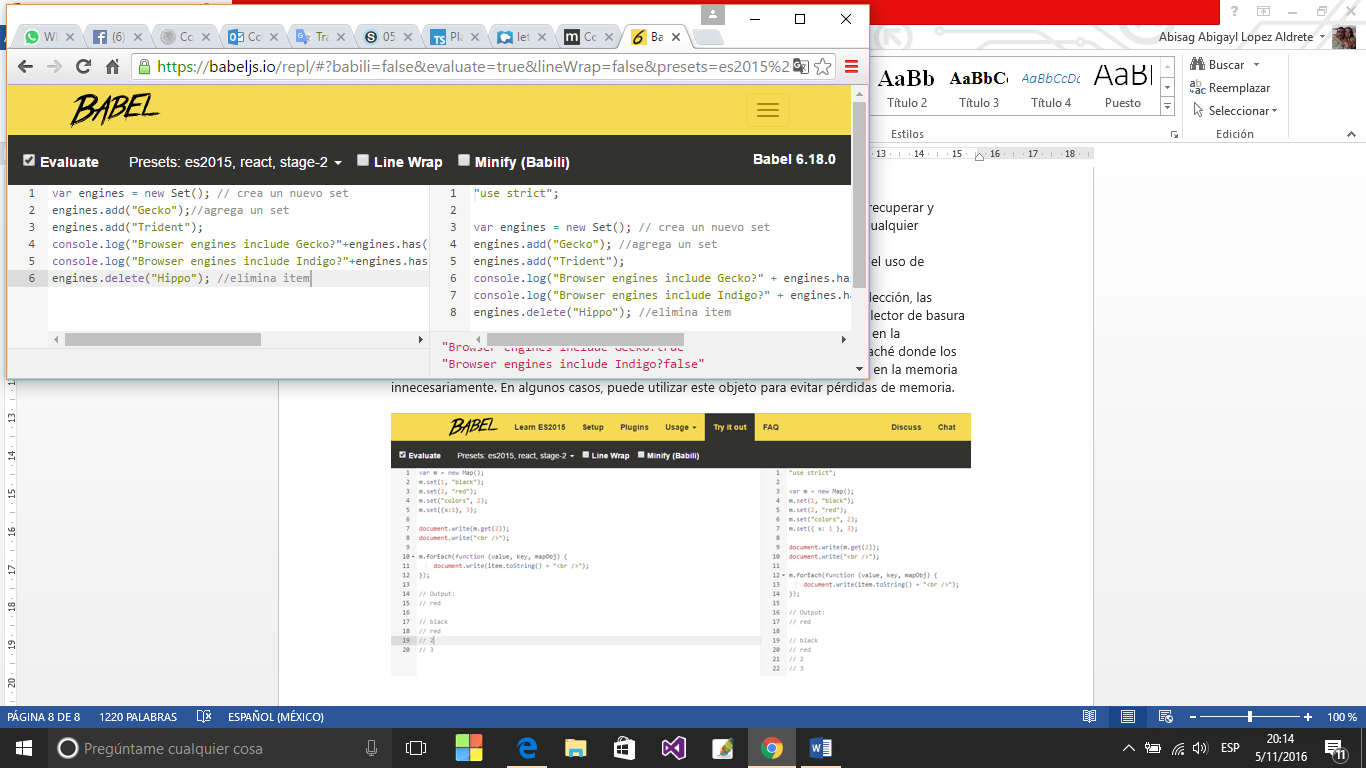
engines.add("Gecko");//agrega un set

engines.add("Trident");

console.log("Browser engines include Gecko?"+engines.has("Gecko")); //verdadero

console.log("Browser engines include Indigo?"+engines.has("Indigo")); //false

engines.delete("Hippo"); //elimina item



Video 8 Iterators and Generators (JavaScript)

Iterators es un objeto que se utiliza para atravesar un objeto contenedor como una lista. En JavaScript, un objeto iterador no es un objeto incorporado distintos, pero es un objeto que implementa una próxima método para acceder al siguiente elemento en el objeto contenedor.

En Microsoft Edge, puede crear sus propios iteradores personalizados. Sin embargo, en general es mucho más fácil de utilizar generadores, que simplifican enormemente la creación de iteradores. Los generadores son un tipo de función que es una fábrica de iteradores. Para crear un iterador personalizado utilizando una función de generador.

La implementación de un iterador JavaScript implica dos o tres objetos que se ajusten a interfaces específicas:

interfaz iterable

interfaz iterador

interfaz IteratorResult

Mediante el uso de estas interfaces, puede crear iteradores personalizados. Esto permite recorrer un objeto iterable usando el de ... de declaración.

interfaz iterable

La interfaz iterable es la interfaz requerida para un objeto iterable (un objeto para el cual se puede obtener un iterador). Por ejemplo, C en para (Sea E de C) debe implementar la interfaz iterable.

JavaScript

obj [Symbol.iterator] = función () { volver iterObj; }

Esta propiedad debe ser una función que no acepte argumentos y devuelve un objeto ( iterObject ) que se ajusta al iterador interfaz.

Muchos tipos incorporados, incluyendo matrices, son ahora iterables. El de ... de bucle consume un objeto iterable. (Sin embargo, no todos los iterables incorporados son iteradores. Por ejemplo, un objeto Array no es un iterador en sí, pero es iterable, mientras que un ArrayIterator también es iterable.)

interfaz iterador

El objeto devuelto por el método Symbol.iterator debe implementar el siguiente método. El siguiente método tiene la siguiente sintaxis.

JavaScript

iterObj.next () = función () { volver iterResultObj; };

El siguiente método es una función que devuelve un valor. La función devuelve un objeto ( iterResultObj ) que se ajusta a la IteratorResult interfaz.Si una llamada previa a la siguiente método de un iterador devuelto un IteratorResult objeto cuyo hecho propiedad es cierto, entonces la iteración se termina y el siguiente método no se llama de nuevo.

Los iteradores también pueden incluir un retorno método para asegurarse de que el repetidor está dispuesto adecuadamente cuando el guión está terminado con ella.

interfaz IteratorResult

La interfaz IteratorResult es el interfaz necesario para el resultado de la siguiente método en un iterador. El objeto devuelto por el próximo debe proporcionar un hecho y el valor de la propiedad.

JavaScript

var iterResultObj = {hecho: verdadero | falso , valor: valor}

El hecho propiedad devuelve el estado de de un iterador siguiente llamada al método, ya sea verdadera o falsa. Si se llega al final del iterador,hecho vuelve verdadera. Si no se alcanza el final, hecho vuelve falsa y un valor está disponible. Si el hecho no existe la propiedad (ya sea por sí mismo o una propiedad heredada), el resultado de hecho se trata como falsa.

Si hecho es falso, el valor de la propiedad devuelve el valor del elemento iteración actual. Si hecho es cierto, este es el valor de retorno del iterador, si se proporciona un valor de retorno. Si el repetidor no tiene un valor de retorno, el valor no está definido. En ese caso, el valor de la propiedad puede estar ausente del objeto conforme si no hereda una propiedad de valor explícito.

generadores

Para crear y utilizar iteradores personalizados fácilmente, crear una función de generador mediante el uso de la sintaxis de la función \* junto con uno o más de rendimiento expresiones. La función devuelve un iterador generador (es decir, un generador), que permite que el cuerpo de la función del generador a ejecutar. La función se ejecuta con el siguiente rendimiento o retorno comunicado.

Llamar al siguiente método de la iterador para devolver el siguiente valor de la función de generador.

El siguiente ejemplo muestra un generador que devuelve un iterador para un objeto de cadena.

JavaScript

función \* stringIter () {

var str = "bobsyouruncle" ;

var idx = 0;

mientras que (IDX <str.length)

str rendimiento [idx ++];

}

var si = stringIter ();

console.log (. si.next () valor);

console.log (. si.next () valor);

console.log (. si.next () valor);

// Salida:

// B

// o

// b

Video9: Array Comprehension

permite ensamblar rápidamente un nuevo arreglo basado en otro existente. Comprensiones de arreglo o listas (o arreglos/listas por comprensión)

Sintaxis

[for (x of iterable) x]

[for (x of iterable) if (condition) x]

[for (x of iterable) for (y of iterable) x + y]

Descripción

Dentro de las comprensiones de arreglo son permitidos estos dos tipos de componentes:

[for...of](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Statements/for...of)

[if](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Statements/if...else)

La iteración for-of es siempre el primer componente. Son permitidas múltiples iteraciones for-of o sentencias if.

Ejemplo

Comprensiones de arreglo simples

[for (i of [ 1, 2, 3 ]) i\*i ];

// [ 1, 4, 9 ]

var abc = [ "A", "B", "C" ];

[for (letters of abc) letters.toLowerCase()];

// [ "a", "b", "c" ]

var temperature = [0,37,100];

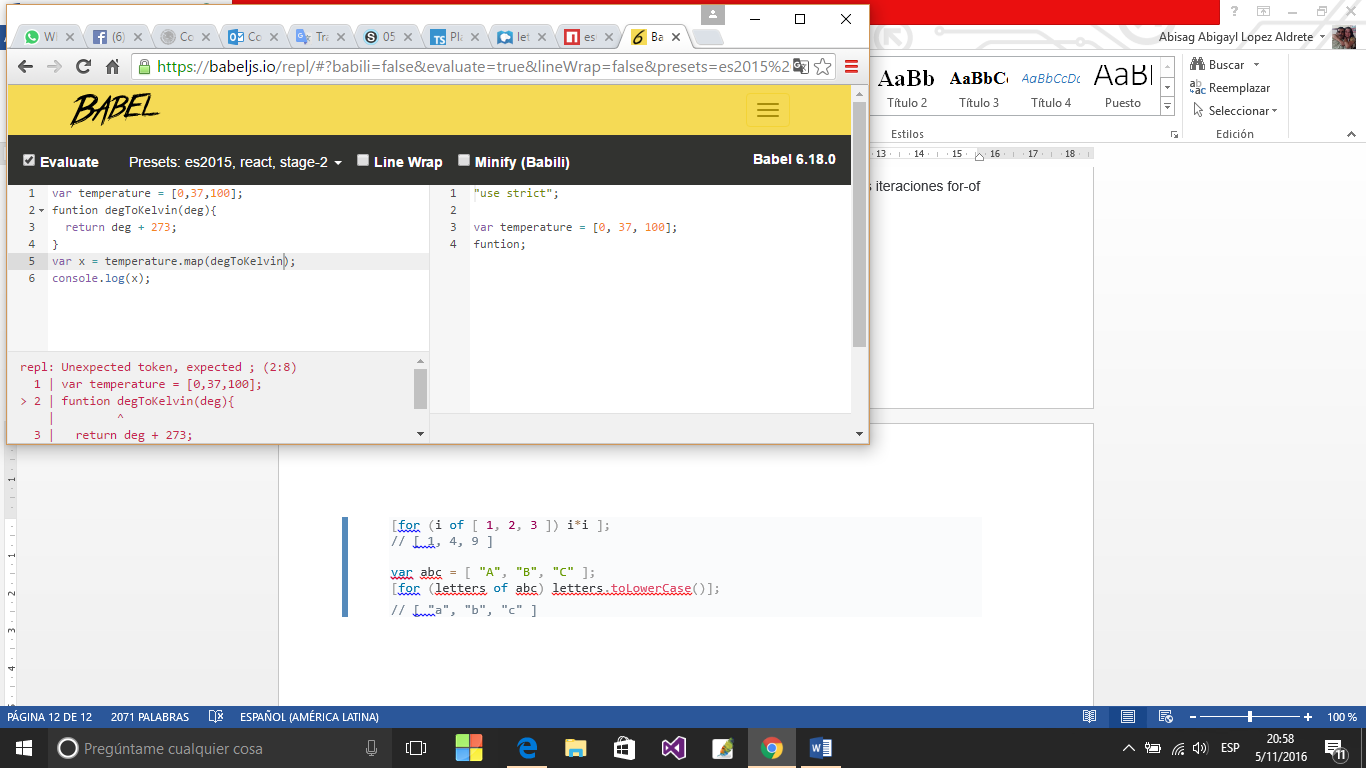
funtion degToKelvin(deg){

return deg + 273;

}

var x = temperature.map(degToKelvin);

console.log(x);



# Video 10 Modules

Las aplicaciones en Angular siguen la estructura modular. Las aplicaciones Angulares contendrán muchos módulos, cada uno dedicado al propósito único. Normalmente, el módulo es un grupo cohesivo de código que se integra con los otros módulos para ejecutar sus aplicaciones angulares.

Un módulo exporta algunas clases, funciones y valores de su código. El Componente es un bloque fundamental de Angular y múltiples componentes conformarán su aplicación.

Un módulo puede ser una biblioteca para otro módulo. Por ejemplo, la biblioteca angular2 / core que es un módulo de biblioteca Angular primario será importada por otro componente.

Por ejemplo, tengo este código:

**Modelos / users.ts**

export class User {

id: number;

name: string; // I want this to be optional

}

**Modelos / mock-users.ts**

import {User} from './user';

export var USERS: User[] = [

{

id: 1

// no name (I wanted it to be optional for user with id 1)

},

{

id: 2,

name: "User 2"

},

]

**servicios / user.service.ts**

import {Injectable} from 'angular2/core';

import {USERS} from './../models/mock-users';

@Injectable()

export class UserService {

getUsers() {

return Promise.resolve(USERS);

}

}

**views / my-component.component.ts**

// imports here...

@Component({

// ...

})

export class MyComponent {

constructor(private \_userService: UserService) { }

getUsers() {

this.\_userService.getUsers().then(users => console.log(users));

}

}

# Video 11 arrow Function or fat arrow

 Arrow function or fat arrow, tiene una sintaxis más corta en comparación con las expresiones de función y léxico se une el thisvalor.Funciones de dirección son siempre en el anonimato.

Sintaxis

([Param] [, param]) => {

declaraciones

}

param => expresión

Ejemplos detallados de sintaxis se pueden ver [aquí](http://wiki.ecmascript.org/doku.php?id=harmony:arrow_function_syntax) .

param

El nombre de un argumento. Cero argumentos deben ser indicados con (). Por sólo un argumento no se requieren los paréntesis. (como foo => 1)

statements or expression

declaraciones múltiples necesitan ser incluido entre llaves, {}. Una sola expresión, sin embargo, no requiere llaves. La expresión es también el valor de retorno implícita de esa función.

Descripción

Dos factores influyeron en la introducción de las funciones de dirección: funciones más cortos y léxica esto.

funciones más cortos

En algunos patrones funcionales, funciones más cortas son bienvenidos. Comparar:

var a = [

"Hidrógeno",

"Helio",

"Litio",

"Berilio"

];

var = a2 a.map (function (s) {return s.length});

var a3 = a.map (s => s.length);

Léxico this

Hasta las funciones de dirección, cada nueva función definida por su propio [este](http://www-db.deis.unibo.it/courses/TW/DOCS/JS/developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this.html) valor (un nuevo objeto en caso de un constructor, no definido en estrictos llamadas a funciones modo, el objeto de contexto si la función se denomina como un "método de objeto", etc.). Esto resultó ser molesto con un estilo orientado a objetos de programación.

Persona function () {

// La Persona () `constructora define this`como ella misma.

  this.age = 0;

setInterval (función growup () {

// En el modo no estricto, la growup () define la función `this`

// Como el objeto global, que es diferente de la `this`

// Definida por el constructor Persona ().

  this.age ++;

}, 1000);

}

var p = new Persona ();

// ES6

function Person() {

this.age = 0;

setInterval(() =>{

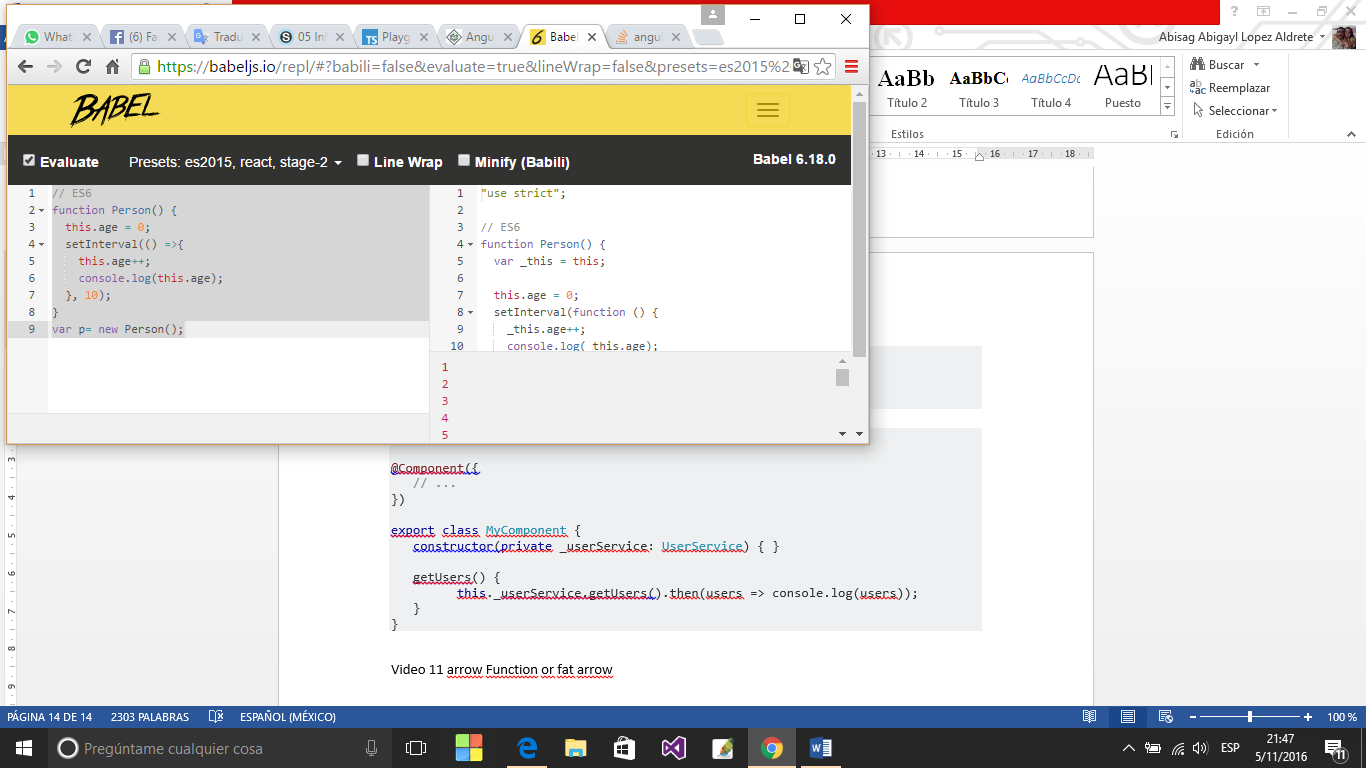
this.age++;

console.log(this.age);

}, 10);

}

var p= new Person();}



# Video 12 Introducción TypeScript

Que es TypeScript?

* Es un lenguaje que compila o crea el código de javascript, mediante unas instrucciones internas
* Cualquier navegador, cualquier host, cualquier sistema operativo, cualquier código fuente.
* Es una tegnologia de Microsoft
* Tu puedes combinar javascript con typescript.

Porque TypeScript?

* Por la necesidad de herramientas de grandes proyectos javascripts
* Arquitectura de aplicaciones javascript de gran escala
* Es una opción
* Tiene clases, herencias, interfaces y mas…

CONFIGURACION DEL AMBIENTE (SETTING UP THE ENVIRONMENT)

* Instalar Node.js : https: //nodejs.org/en/
* Ir a la pagina oficial de Typescript: <http://www.typescriptlang.org/>
* Abrir la terminal y executar:
  + Sudo npm install –g typescript
* Creat un archivo javascript con el nombre :Holamundo.ts
  + Tsc holamundo.ts (en la terminal node.js command prompt, para crear el archivo javascript.

# Video 13 system part 1y 2

Porque usar Type Systems?

Porque el type systems hace el código mas fácil para mantener utilizar

* Puede hacer que el código sea más legible
* Puede facilitar el análisis del código
* Puede permitir refactorización fiable
* Puede permitir un mejor soporte IDE en general
* Puede detectar errores temprano

EL CASO MÁS IMPORTANTE DEL USO

Estructuras de datos de estilo JSON

* carga útil REST
* archivos de configuración
* Objetos desde y hacia la base de datos
* Como un segundo lugar cercano: encontrar usos

TYPESCRIPT

EASE OF USE AND TOOL SUPPORT OVER SOUNDNESS

http://www.typescriptlang.org/

By Microsoft (Anders Hejlsberg)

Based on ES6 (probably ES7/ES8)

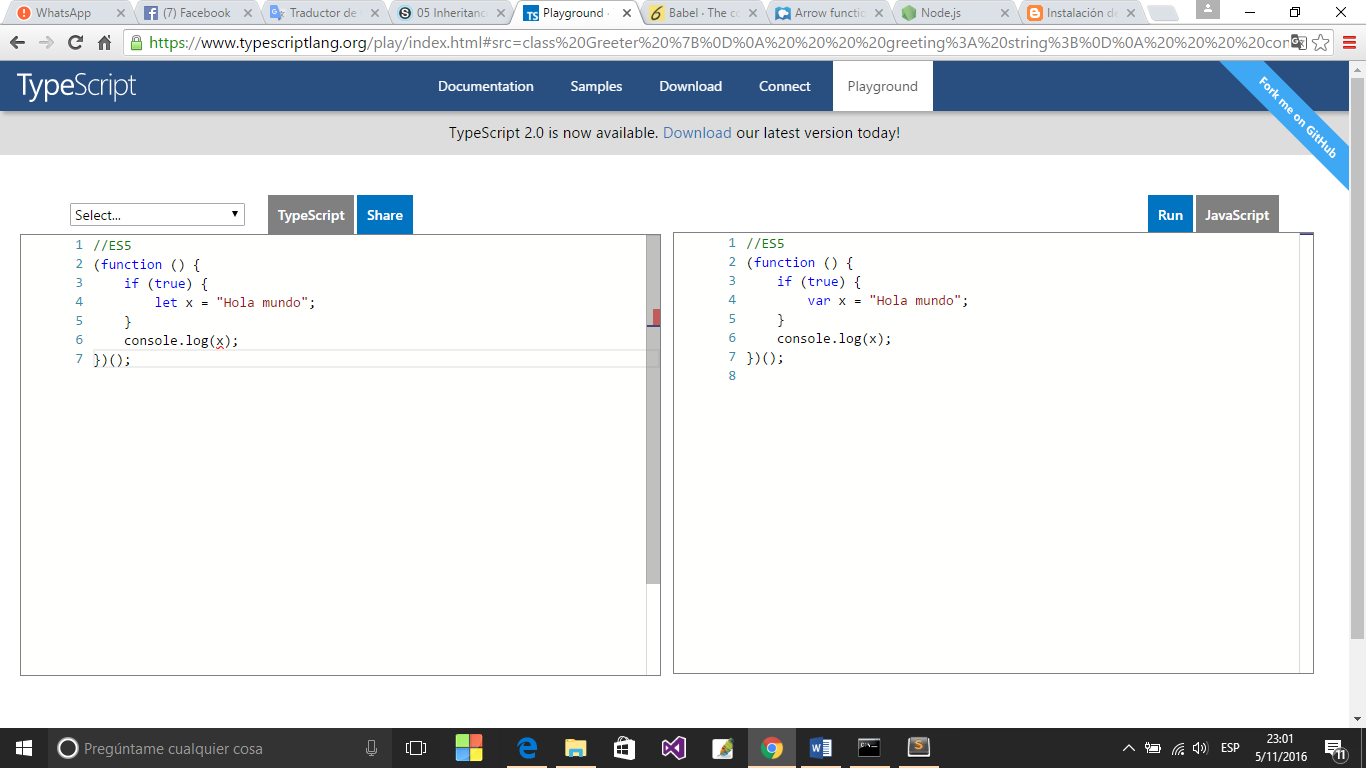
Adds optional type annotations, visibility, and decorators

Compiler checks and removes annotations

Latest 1.8 release adds more general sane checks

External declarations can add type information to pure JavaScript

Extensive support in WebStorm and Visual Studio Code



# Video 15. Funciones



Se definen como **un grupo o conjunto de sentencias que solucionan un problema particular**.

Tanto en Javascript como en TypeScript, las funciones tienen una cabecera (donde se define el nombre de la función) y un cuerpo (las instrucciones). Pueden ser definidas de diferentes formas, aunque en todos :

* Funciones con nombre
* Funciones anónimas o métodos anónimos
* Funciones lambda
* Funciones definidas en las clases

Todo un ramillete de opciones, para empezar. En todos los casos, las funciones aceptan argumentos, que además se puede forzar a que correspondan a cierto tipo. Pero ¿que pasa cuando queremos un número no determinado de argumentos? En estos casos, se utiliza ? tras el último parámetro.

Otra circunstancia interesante es la posibilidad de asignar valores predeterminados a los parámetros (cómo en Python), simplemente asignando el valor en la definición. También es interesante el uso de **…variable: string[]**, que es la forma de permitir un número indeterminado de parámetros, que acaban recogidos en la variable *array* **variable**.

|  |  |
| --- | --- |
| 1  2  3  4  5 | function buildName(firstName: string, ...restOfName: string[]) {  return firstName + " " + restOfName.join(" ");  }    var foo = buildName("Joseph", "Samuel", "Lucas"); var foo = buildName("Maria", "Lu"); |

Cómo veis, se cubren todo tipo de opciones respecto a los argumento de las funciones.

Y ahora, empezamos con las funciones

**Funciones con nombre**

La primera de las opciones se diferencia de las funciones con nombre de Javascript en que los argumentos pueden asignarse tipos, y lo que devuelven las funciones también

|  |  |
| --- | --- |
| 1 | function foo(msg: string) { console.log(msg); } |

Pocas diferencias respecto a esta misma función si se escribiera directamente en Javascript:

|  |  |
| --- | --- |
| 1 | function foo(msg) { console.log(msg); } |

**Funciones anónimas**

Quizás, esta forma de crear funciones es la que yo más utilizo en Javascript, y al igual que antes, la diferencia es que pueden asignarse tipos en los argumentos y en el resultado:

|  |  |
| --- | --- |
| 1 | var foo = function(x: number, y: number): number { return x+y; } |

En Javascript, es muy similar:

|  |  |
| --- | --- |
| 1 | var foo = function(x, y) { return x+y; } |

Si la función no devuelve nada, puede marcarse como void, esto es, sin devolución (cómo en Java o en C#).

**Funciones lambda, o funciones de flechas gordas**

Las funciones lambda son funciones de una sola instrucción (normalmente) que se almacenan en una variable, o que se ejecutan directamente. Se caracterizan de que NO hace falta escribir return, porque se sobre entiende que la función devuelve algo. Estas funciones lambda no son exclusivas de TypeScript, pues lenguajes como C# o Python también las tienen.

En TypeScript, es identifican por **=>**, que hace de separador entre los argumentos, y lo que devuelve:

|  |  |
| --- | --- |
| 1 | var foo = (x: number, y: number) => x+y; |

En realidad, la función sin lambda sería tal que así:

|  |  |
| --- | --- |
| 1 | var foo = function (x, y) { return x + y; }; |

En este caso, se trata de ofrecer diferentes formas de escribir el mismo código a los desarrolladores, algo que a mi, personalmente, me parece lioso. Pero, es mi opinión personal.

**Funciones en las clases**

Cuando se definen funciones en las clases, una de sus peculiaridades es que no aparece function, sino que se escriben directamente:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class NewClass {    foo(x: number, y:number): number { return x+y; }    } |

**Ejemplo de funciones en TypeScript**

Es hora de aplicar lo aprendido, y vamos con un ejemplo. Se trata de preparar un pequeño formulario, donde poner dos números o cadenas, y sumarlos.  Aquí solo pondré el código TypeScript, pues es el que nos interesa.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | (function () {        //Lambda function      var $ = (id) => document.getElementById(id);        var totalButton: HTMLButtonElement = <HTMLButtonElement>$('sumar');      var uno: HTMLInputElement = <HTMLInputElement>$('uno');      var dos: HTMLInputElement = <HTMLInputElement>$('dos');      var resultado = $('resultado');        var ejecuta = function(): void {          console.log(uno.value);          console.log(dos.value);          resultado.innerHTML = uno.value + dos.value;      }      totalButton.addEventListener('click', (e) => {          ejecuta()      })      uno.addEventListener('change', (e) => {          // resultado.innerHTML = dos.value + uno.value;          ejecuta();      })    })(); |

# Video 16. Clases, objets, interfaces and inheritance

**objeto y interfaz**

Uno de los principios fundamentales del texto mecanografiado es que la comprobación de tipos se centra en la formaque tienen los valores. A veces se denomina "tipificación de pato" o "subtipos estructural". A máquina, las interfaces llenan el papel de nombrar a estos tipos, y son una poderosa manera de definir los contratos dentro de su código, así como los contratos con el código fuera de su proyecto.

Nuestra primera interfaz

La forma más fácil de ver cómo funcionan las interfaces es comenzar con un ejemplo sencillo:

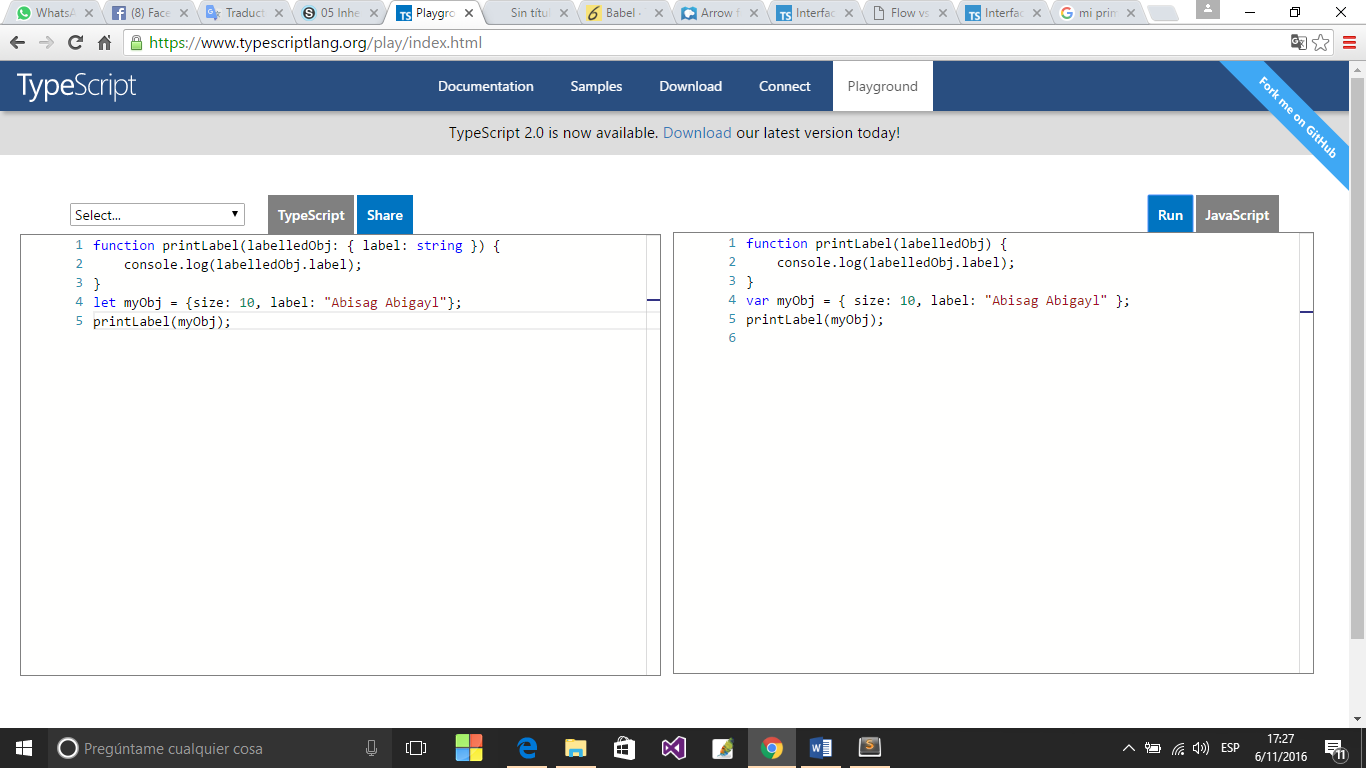
function printLabel(labelledObj: { label: string }) {

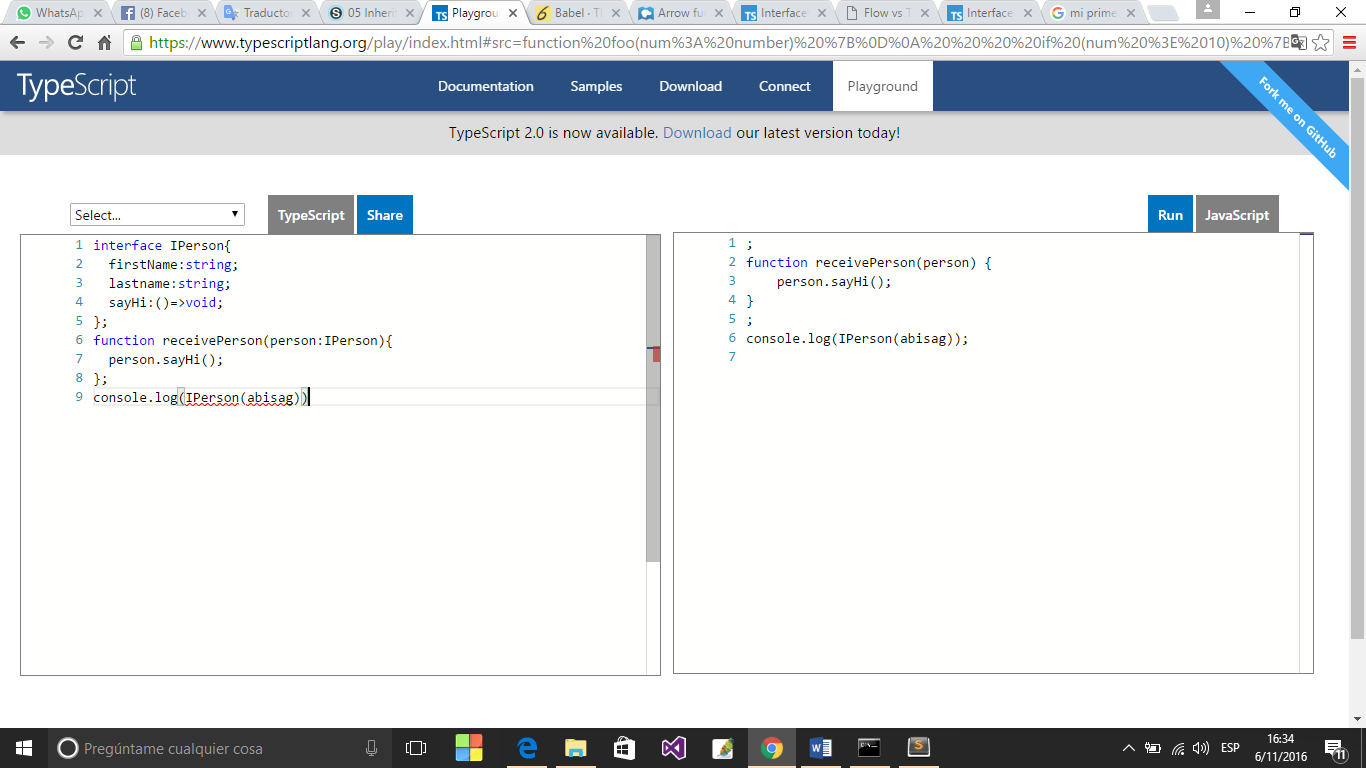
console.log(labelledObj.label);

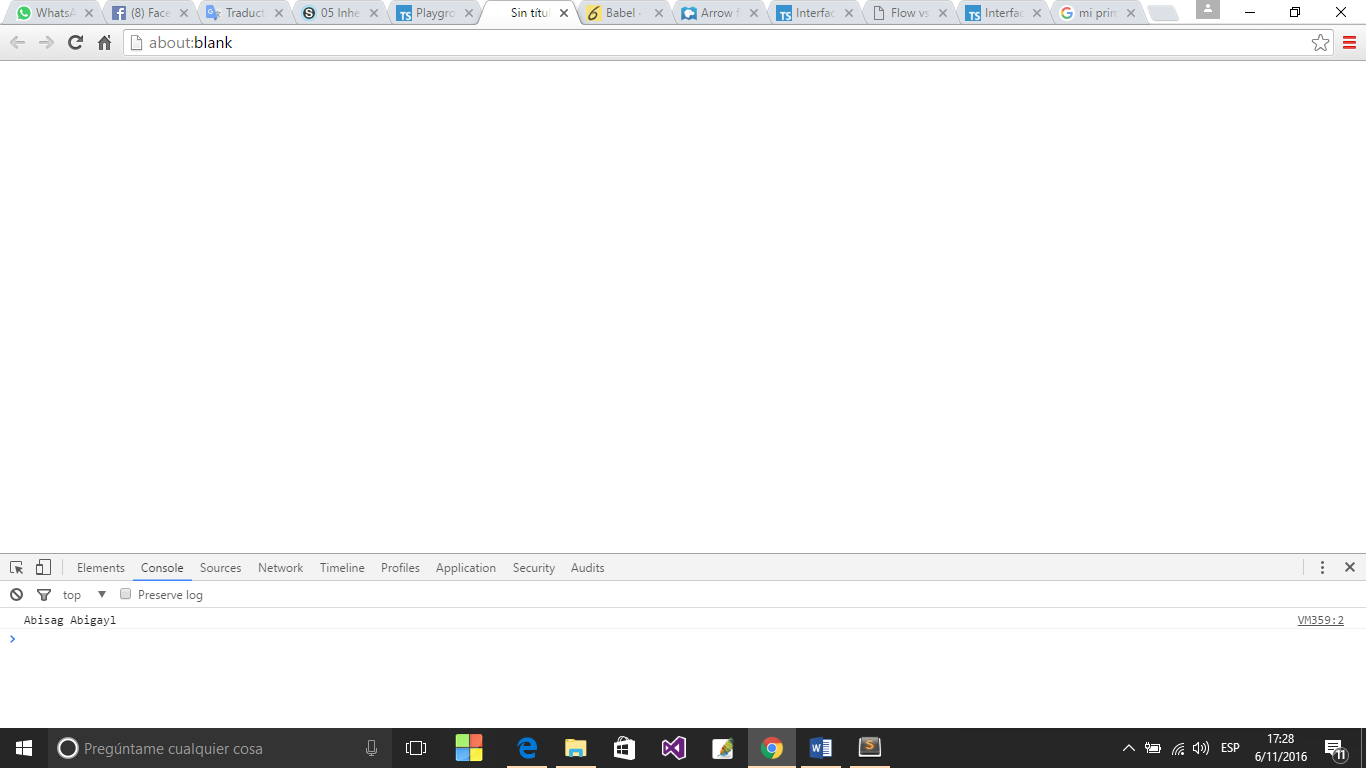
}

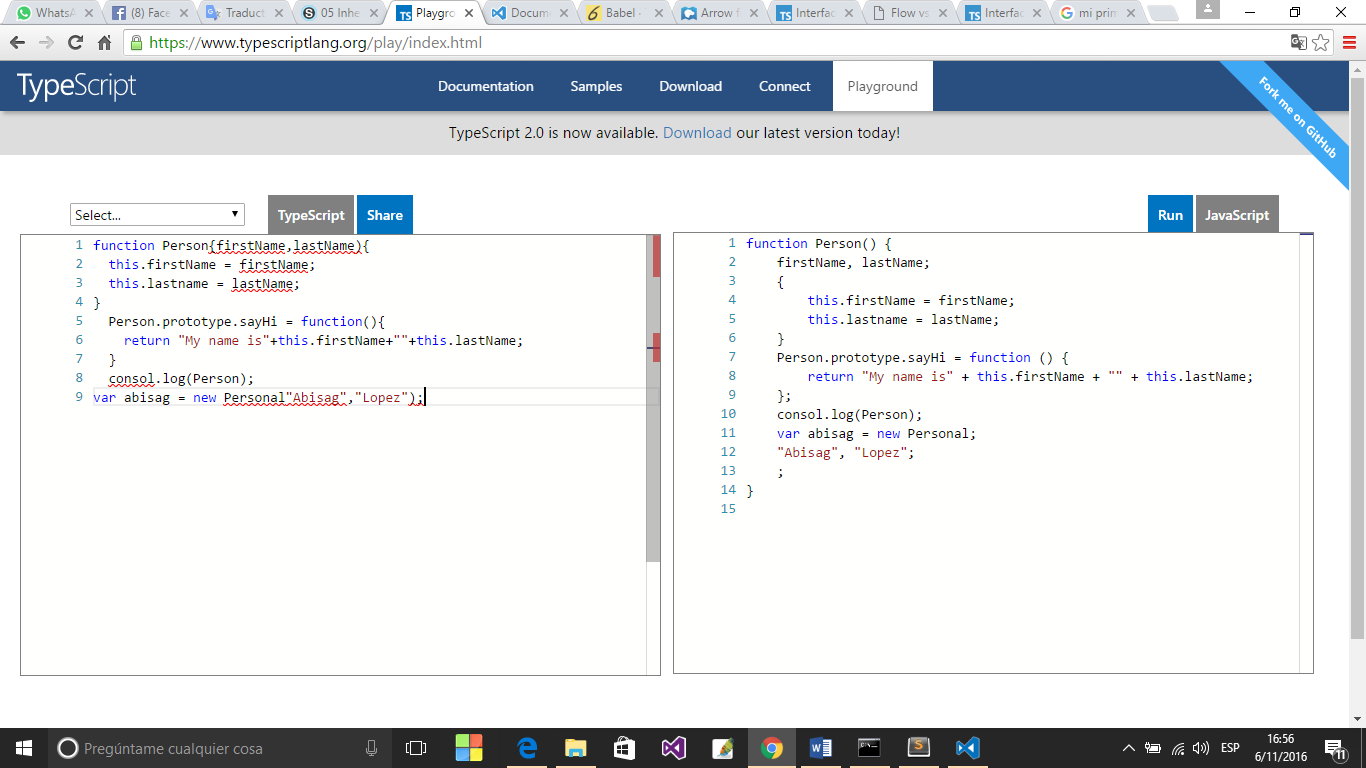
let myObj = {size: 10, label: "Abisag Abigayl"};

printLabel(myObj);









interface IPerson{

firstName; string;

lastName: string;

}

class Person implements IPerson{

firstName; string;

lastName: string;

constructor(firstName:string,lastName:string = "Abisag"){

this.firstName = firstName;

this.lastName = lastName;

}

getfullName():string {

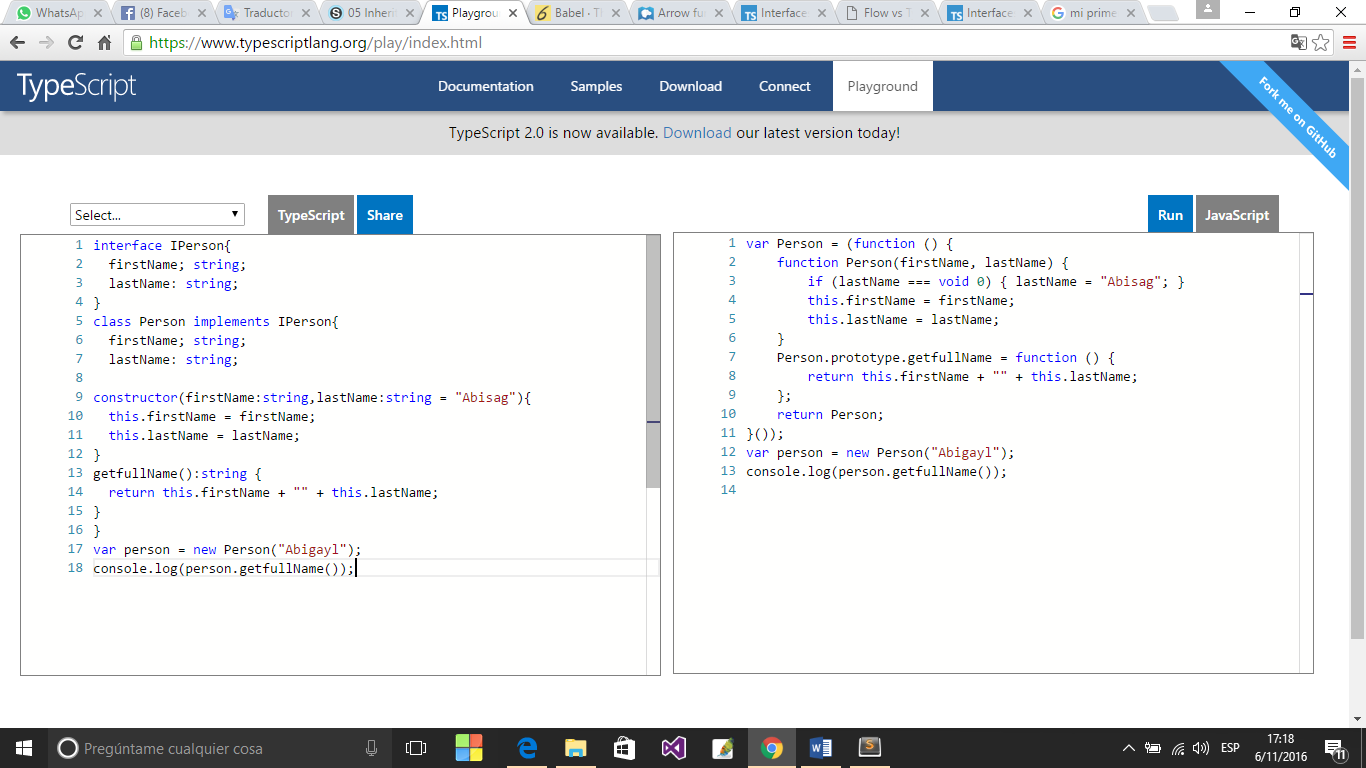
return this.firstName + "" + this.lastName;

}

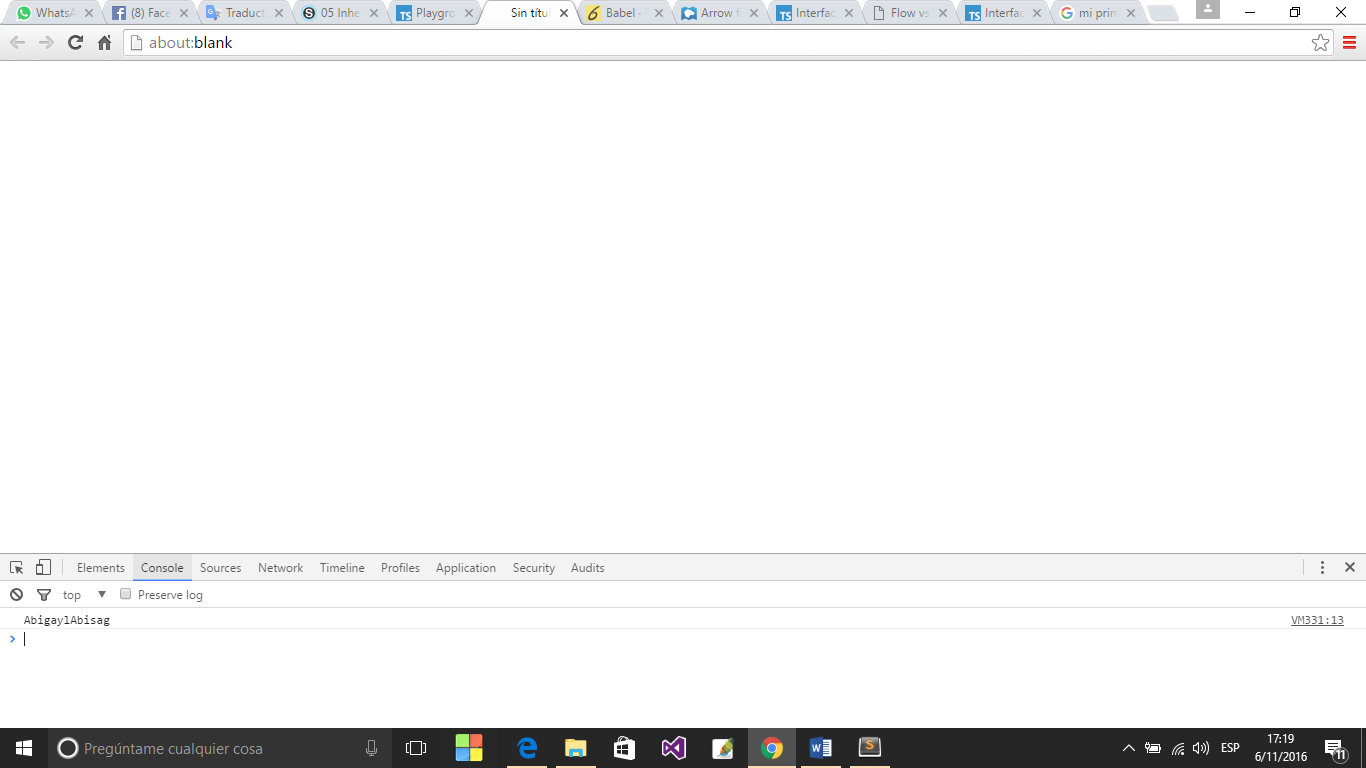
}

var person = new Person("Abigayl");

console.log(person.getfullName());



Se pone run, abre una pagina nueva, se le da clic derecho, inspeccionar, console



# Video 17 classes, objets interfaces and inheritace parte 2

inheritace

Los programadores de JavaScript serán capaces de construir sus aplicaciones el uso de este enfoque basado en la clase orientada a objetos. En typescript, dejamos que los desarrolladores utilizar estas técnicas ahora, y compilarlas en JavaScript que funciona en todos los navegadores y plataformas, sin tener que esperar a la próxima versión de JavaScript.

clases

Vamos a echar un vistazo a un ejemplo sencillo basado en clases:

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {

return "Hello, " + this.greeting;

}

}

let greeter = new Greeter("world");

La sintaxis debe resultar familiar si has utilizado C # o Java antes. Declaramos una nueva clase Greeter. Esta clase tiene tres miembros: una propiedad llamada greeting, un constructor y un método greet.

Se dará cuenta de que en la clase cuando nos referimos a uno de los miembros de la clase que anteponer this.. Esto denota que es un miembro de acceso.

En la última línea se construye una instancia de la Greeterclase utilizando new. Esto pone en el constructor que definimos anteriormente, la creación de un nuevo objeto con la Greeterforma y el funcionamiento del constructor para inicializarlo.

Herencia

A máquina, podemos utilizar patrones orientados a objetos comunes. Por supuesto, uno de los patrones más fundamentales en la programación basada en la clase es ser capaz de extender las clases existentes para crear otras nuevas mediante herencia.

Vamos a echar un vistazo a un ejemplo:

class Animal {

name: string;

constructor(theName: string) { this.name = theName; }

move(distanceInMeters: number = 0) {

console.log(`${this.name} moved ${distanceInMeters}m.`);

}

}

class Snake extends Animal {

constructor(name: string) { super(name); }

move(distanceInMeters = 5) {

console.log("Slithering...");

super.move(distanceInMeters);

}

}

class Horse extends Animal {

constructor(name: string) { super(name); }

move(distanceInMeters = 45) {

console.log("Galloping...");

super.move(distanceInMeters);

}

}

let sam = new Snake("Sammy the Python");

let tom: Animal = new Horse("Tommy the Palomino");

sam.move();

tom.move(34);

Este ejemplo abarca un buen número de las características de herencia a máquina de escribir que son comunes a otros idiomas. Aquí vemos las extendspalabras clave utilizadas para crear una subclase. Esto se puede ver cuando HorseySnakesubclase de la clase base Animaly tener acceso a sus características.

Las clases derivadas que contienen funciones constructoras deben llamar a super()la cual se ejecutará la función constructora de la clase base.

function foo (num: number){

if(num > 10){

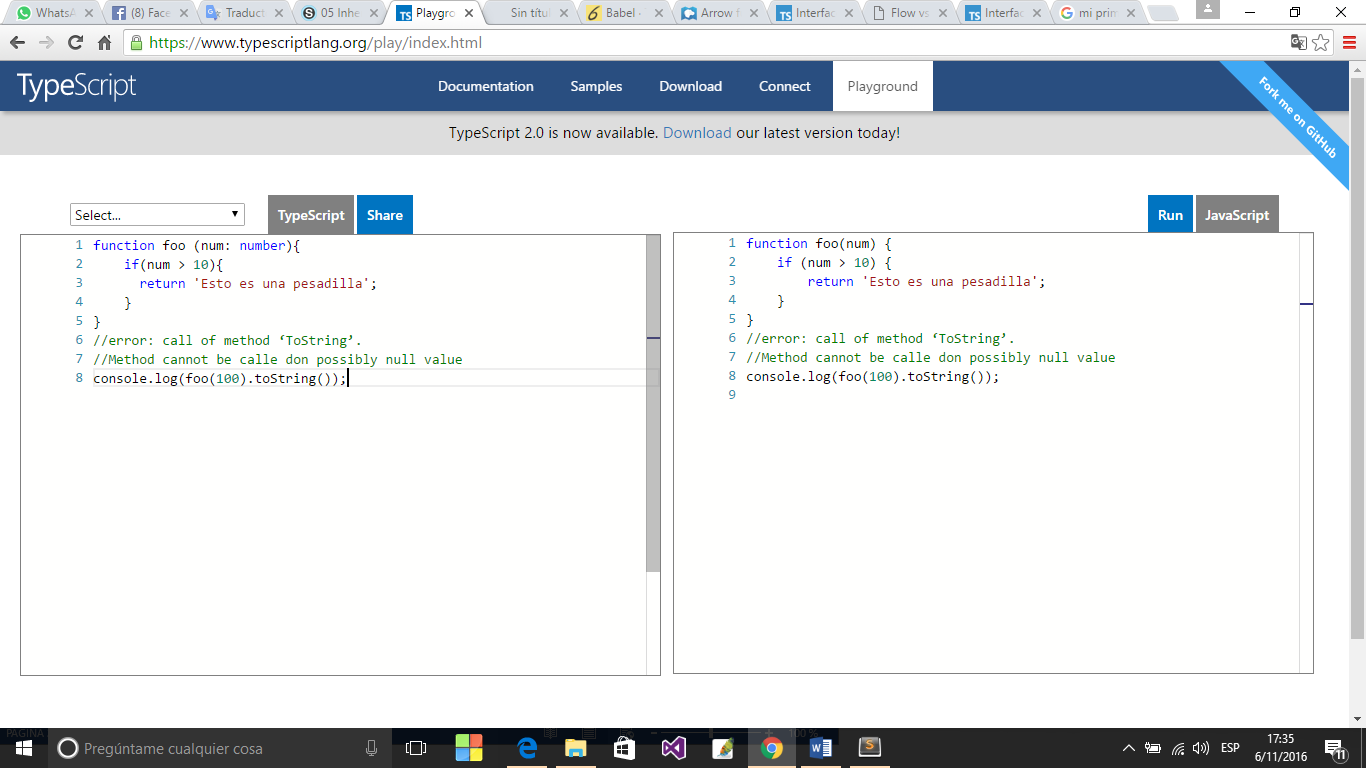
return 'Esto es una pesadilla';

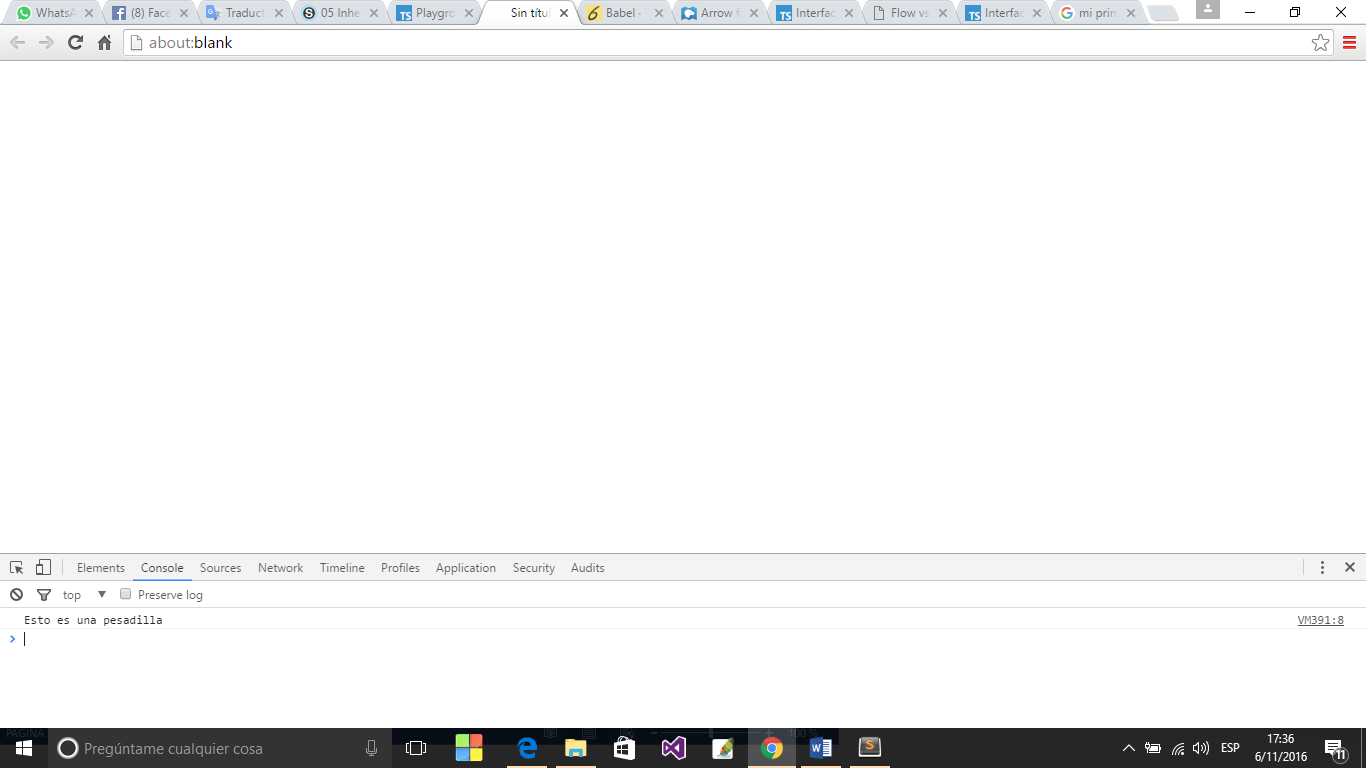
}

}

//error: call of method ‘ToString’.

//Method cannot be calle don possibly null value

console.log(foo(100).toString());



# Video 18 modules

* No usa variables globales
* El código de tu proyecto para otro código javascript
* Los moculos en typescript son:
  + Modulos internos
  + Modulos externos

Los módulos se ejecutan dentro de su propio ámbito de aplicación, no en el ámbito mundial; esto significa que las variables, funciones, clases, etc. declaradas en un módulo no son visibles fuera del módulo, a menos que explícitamente se exportan con una de las [exportformas](https://www.typescriptlang.org/docs/handbook/modules.html#export) . Por el contrario, para consumir una variable, función, clase, interfaz, etc. exportado de un módulo diferente, tiene que ser importados utilizando una de las [importformas](https://www.typescriptlang.org/docs/handbook/modules.html#import) .

Los módulos son declarativa; las relaciones entre los módulos se especifican en términos de importaciones y exportaciones a nivel de archivo.

Módulos de importación entre sí usando un cargador de módulos. En tiempo de ejecución del cargador de módulos es responsable de localizar y ejecutar todas las dependencias de un módulo antes de ejecutarlo. Conocidos módulos de cargadores frontales ocasión en JavaScript son los [CommonJS](https://en.wikipedia.org/wiki/CommonJS) cargador de módulos para Node.js y [require.js](http://requirejs.org/) para aplicaciones web.

En typeScrip, al igual que en ECMAScript 2015, cualquier archivo que contiene un alto nivel import o export se considera un módulo.

**Externo**

Cualquier declaración (como una variable, función, clase, tipo de alias, o interfaz) se puede exportar mediante la adición de laexportpalabra clave.

Validation.ts

**export** **interface** StringValidator {

isAcceptable(s: string): boolean;

}

ZipCodeValidator.ts

**export** **const** numberRegexp = /^[0-9]+$/;

**export** **class** ZipCodeValidator **implements** StringValidator {

isAcceptable(s: string) {

**return** s.length === 5 && numberRegexp.test(s);

}

}

**Export statements**

Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

class ZipCodeValidator implements StringValidator {

isAcceptable(s: string) {

return s.length === 5 && numberRegexp.test(s);

}

}

export { ZipCodeValidator };

export { ZipCodeValidator as mainValidator };

Re-exports

Often modules extend other modules, and partially expose some of their features. A re-export does not import it locally, or introduce a local variable.

ParseIntBasedZipCodeValidator.ts

export class ParseIntBasedZipCodeValidator {

isAcceptable(s: string) {

return s.length === 5 && parseInt(s).toString() === s;

}

}

// Export original validator but rename it

export {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidator";

Optionally, a module can wrap one or more modules and combine all their exports using export \* from "module" syntax.

**AllValidators.ts**

export \* from "./StringValidator"; // exports interface 'StringValidator'

export \* from "./LettersOnlyValidator"; // exports class 'LettersOnlyValidator'

export \* from "./ZipCodeValidator"; // exports class 'ZipCodeValidator'

**Import**

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the import forms below:

Import a single export from a module

import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();

imports can also be renamed

import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";

let myValidator = new ZCV();

Import the entire module into a single variable, and use it to access the module exports

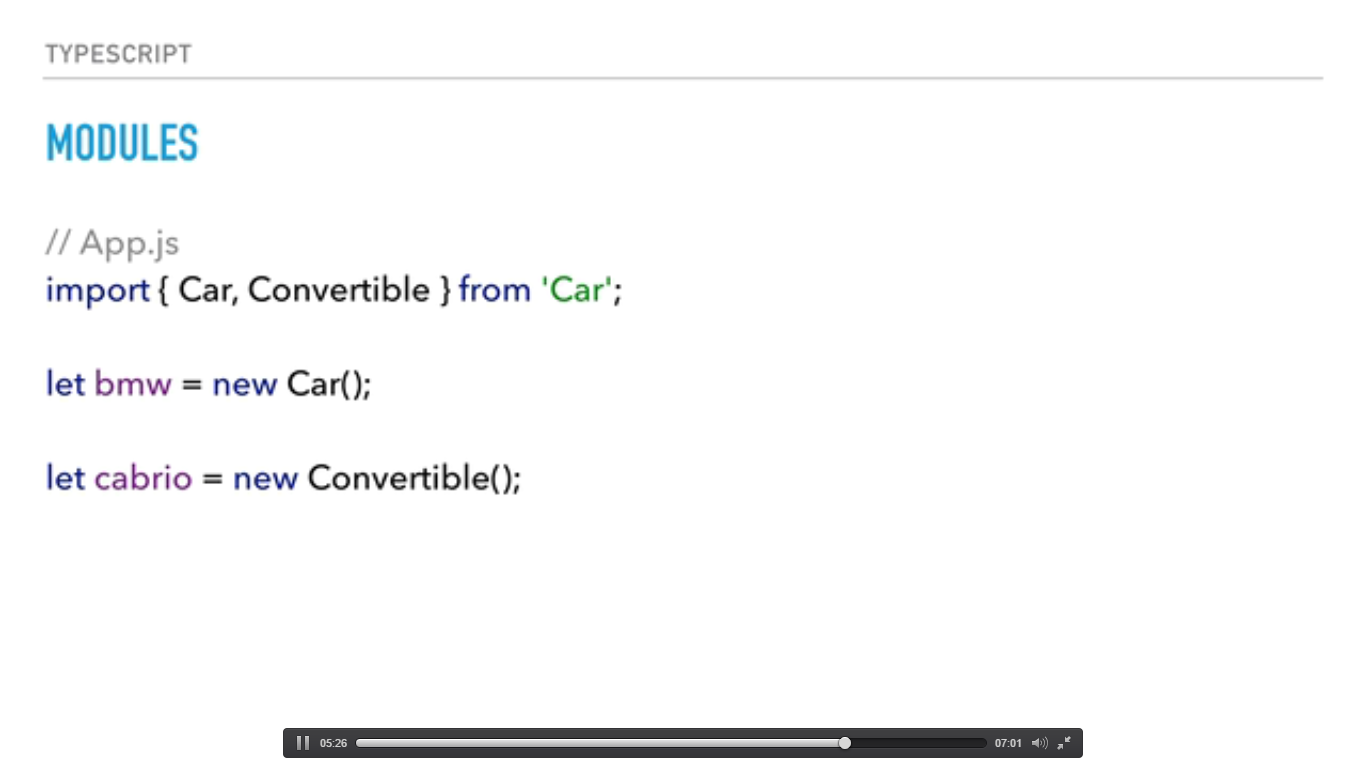
import \* as validator from "./ZipCodeValidator";

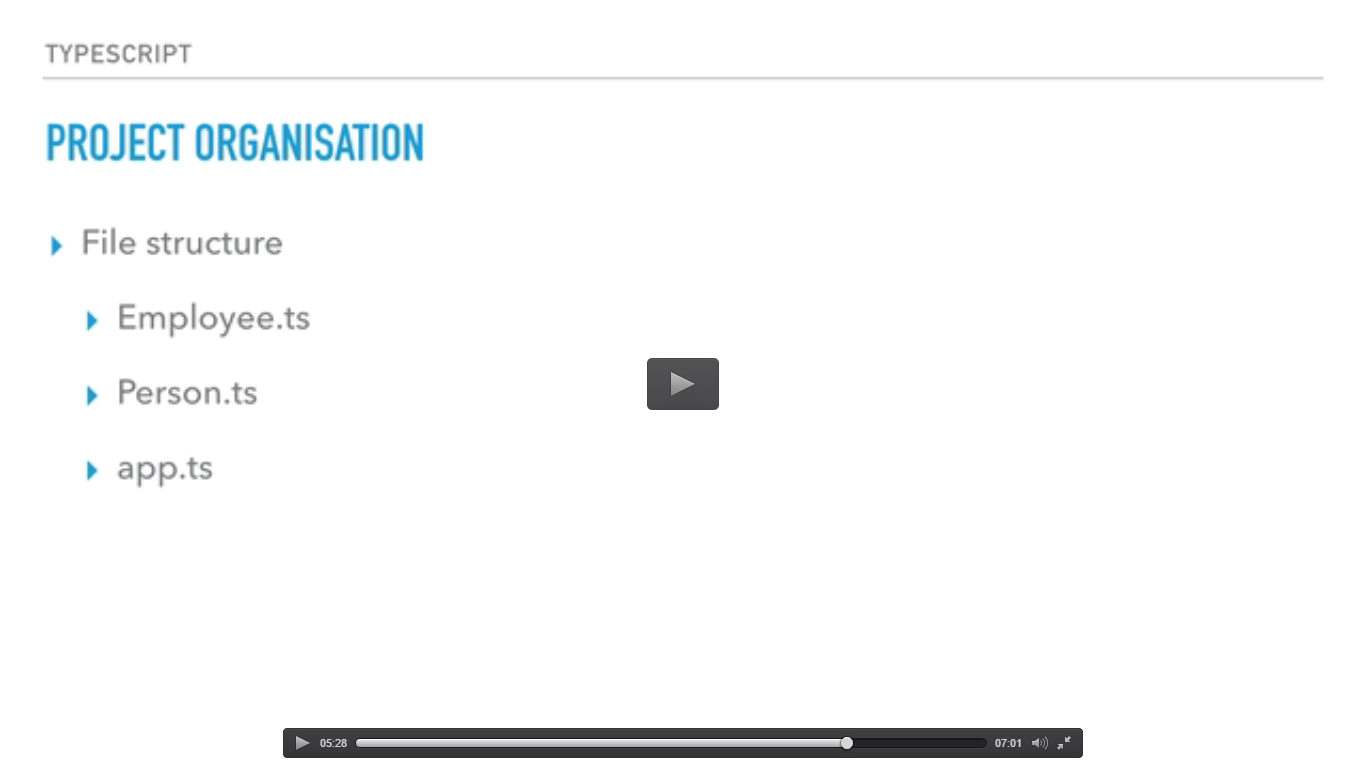
let myValidator = new validator.ZipCodeValidator();

Import a module for side-effects only (Importar un módulo de sólo efectos secundarios)

Aunque la práctica no es recomendable, algunos módulos establecieron un estado global que puede ser utilizado por otros módulos. Estos módulos pueden no tener ninguna exportación, o si el cliente no está interesado en cualquiera de sus exportaciones. Para importar estos módulos, utilice:

import "./my-module.js";



# Video 19. Decorators

Un decorador es una expresión que se evalúa después de que se ha definido una clase, que se puede utilizar para anotar o modificar la clase de alguna manera.

