TypeScript / JavaScript

TypeScript

- Desarrollador por Microsoft.

- Es un super conjunto de JavaScript: Cualquier código válido en JavaScript es válido en TypeScript.

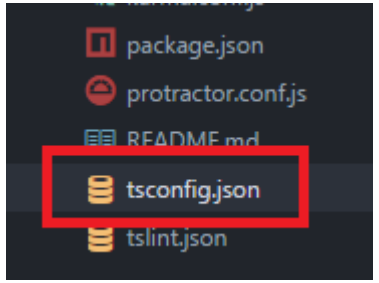
- El código es compilado a JavaScript.

- Ayuda a cometer la menor cantidad de errores posible en nuestro código. errores comunes al momento de escribir código:
  - Una variable no estaba definida.
  - Propiedades que no tiene un objeto.
  - Sobre escritura de variables, métodos, constantes.
  - Problemas de los que generalmente nos damos cuenta hasta que el código está en ejecución.

- El código es más fácil de entender.

TypeScript

tsconfig.json




```
{
  "compileOnSave": false,
  "compilerOptions": {
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2017",
      "dom"
    ]
  }
}
```

Archivo de configuración del compilador de TypeScript.

- Generado ejecutando el comando “tsc --init”

- sourceMap: Habilita el debugger de archivos TypeScript durante la ejecución.

- declaration: Definir implementaciones que deben estar disponibles a nivel de compilación.

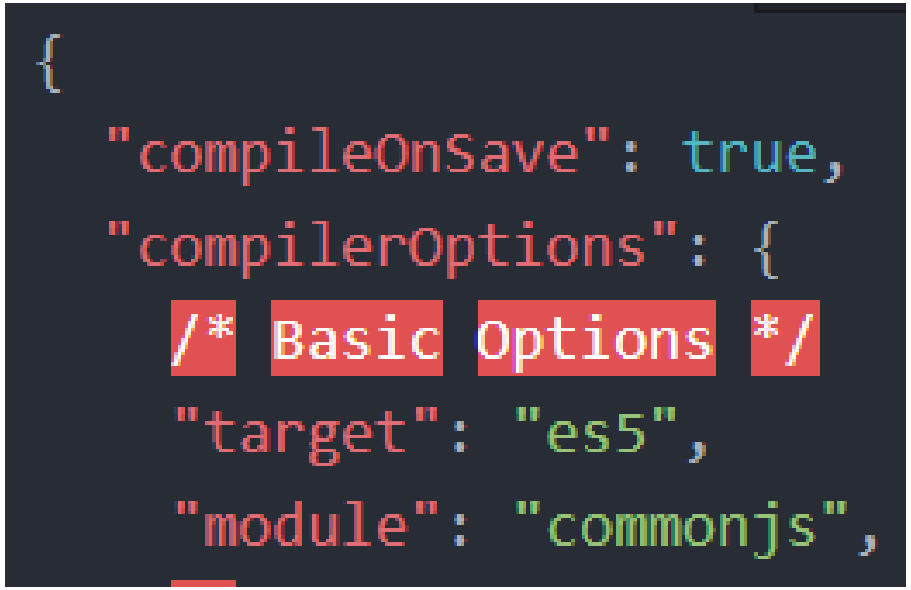
- moduleResolution: Indica al compilador en dónde se encuentran los módulos. El valor “node” indica que se va utilizar la carpeta node_modules.

- target: Versión de JavaScript (ECMA Script) a la que queremos compilar.

TypeScript

tsconfig.json

Para que el proyecto compile de forma automática los cambios realizados al código, hay que modificar el archivo tsconfig.json incluyendo la propiedad “compileOnSave”:true:



TypeScript

tsconfig.json

Adicionalmente, vamos a indicar que se utilizará la versión ECMA Script 2019:

```
{
  "compileOnSave": true,
  "compilerOptions": {
    /* Basic Options */
    // "incremental": true,
    "target": "ES2019",
    "module": "commonjs",
```

TypeScript

var Vs. let

```
var numero1:number = 1;
var numero2:number = 2;

if(true){
  var numero1:number = 3;
  var numero2:number = 4;
  console.log("suma if = " + (numero1 + numero2));
}

console.log("suma = " + (numero1 + numero2));
```

Vs.

```
let numero1:number = 1;
let numero2:number = 2;

if(true){
  let numero1:number = 3;
  let numero2:number = 4;
  console.log("suma if = " + (numero1 + numero2));
}

console.log("suma = " + (numero1 + numero2));
```

¿Cuál es el resultado en cada caso?

TypeScript

Tipos de Datos

```typescript
let texto:string = "Hola Mundo";  //Variable de tipo texto
let numero:number = 9876;         //Variable de tipo numerico
let booleano:boolean = false;     //Variable de tipo booleano
let fecha:Date = new Date();      //Variable de tipo fecha
let flexible:any;                 //Variable que puede almacenar cualquier tipo de dato
flexible = "Hola Mundo";
flexible = 1234;
flexible = false;
flexible = new Date;

let persona = {                   //Variable de tipo objeto
  nombre: "Jorge",
  apellido: "Miramontes"
}
```

TypeScript

Template Literals

Se define con el caracter comilla simple invertida: ``

```typescript
let nombre:string = "Jorge";
let apellido:string = "Miramontes";
let dato:string = "valor";

let formaTradicional = "Nombre: " + nombre + "\n" + "Apellido: " + apellido + "\n" + "Dato: " + dato;

let templateLiteral = `Nombre: ${nombre}
Apellido: ${apellido}
Dato: ${dato}`;

let templateLiteralTabs = `Nombre: ${nombre}
                          Apellido: ${apellido}
                          Dato: ${dato}`;

console.log(formaTradicional);
console.log(templateLiteral);
console.log(templateLiteralTabs);

function suma(numero1:number, numero2:number){
  return numero1 + numero2;
}

let sumaDirecta = `${1 + 2}`;
let sumaFuncion = `${suma(1,2)}`;
console.log(sumaDirecta);
console.log(sumaFuncion);
```

TypeScript

Funciones

¿Cuál es el resultado?

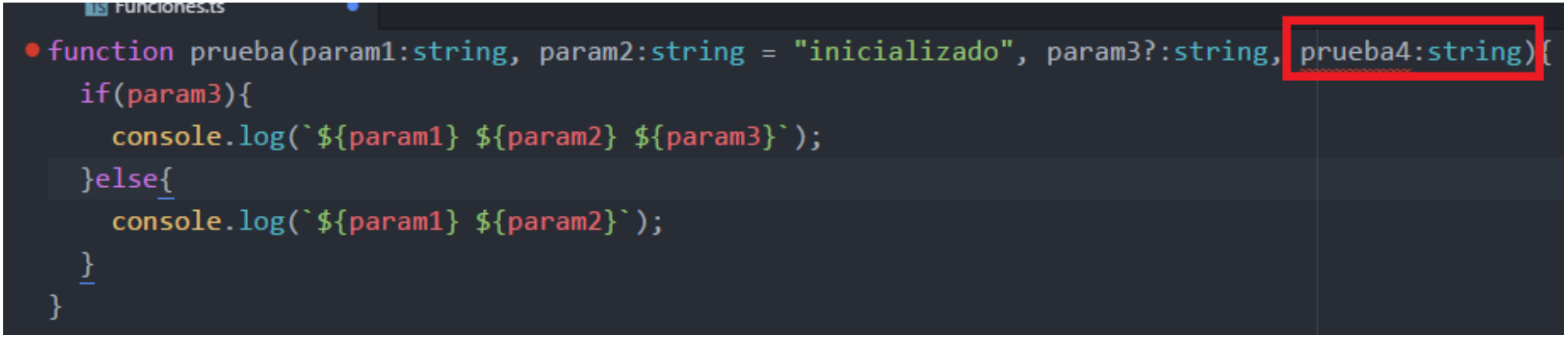
```
function prueba(param1:string, param2:string = "inicializado", param3?:string){
  if(param3){
    console.log(`${param1} ${param2} ${param3}`);
  }else{
    console.log(`${param1} ${param2}`);
  }
}

prueba("param1");
prueba("param1","param2");
prueba("param1","param2","param3");
```

TypeScript

Funciones

No se pueden poner parámetros obligatorios al final. Todos los parámetros obligatorios debe de indicarse al inicio.



TypeScript

Funciones de Flecha

Sintaxis:

(parametros) => { operaciones y retorno }

```
let operacionNormal = function suma(numero1:number, numero2:number){
  console.log(`En Funcion Normal. numero1: ${numero1}, numero2: ${numero2}`);
  return numero1 + numero2;
}
```

Función Normal

Arrow Function

```
let operacionArrowFunction = (numero1:number, numero2:number) => {
  console.log(`En Arrow Function. numero1: ${numero1}, numero2: ${numero2}`);
  return numero1 + numero2;
}
```

```
console.log(operacionNormal(1,2));
console.log(operacionFuncionFlecha(1,2));
```

TypeScript

Funciones de Flecha

Alcance de “this”:

```
let persona1 ={
  nombre1:"Roger",
  apellidoPaterno1:"Waters",
  habilidad1:"Bajo",
  imprime(){
    setTimeout(function(){
      console.log(`${this.nombre1} ${this.apellidoPaterno1} ${this.habilidad1}`);
    }, 3000)
  }
}
```

Dentro de un contexto asíncrono, una función normal NO tiene referencia a ***this***

```
undefined undefined undefined    funcionFlecha.js:11
```

TypeScript

Funciones de Flecha

Con funciones flecha:

```
let persona2 ={
  nombre1:"Roger",
  apellidoPaterno1:"Waters",
  habilidad1:"Bajo",
  imprime(){
    setTimeout(() => {
      console.log(`${this.nombre1} ${this.apellidoPaterno1} ${this.habilidad1}`);
    }, 3000)
  }
}
```

Dentro de un contexto asíncrono, una función flecha SÍ tiene referencia a ***this***

```
Roger Waters Bajo                                    ArrowFunctions.js:42
```

TypeScript

Funciones de Flecha

```
let persona={
  dato1:"d1",
  dato2:"d2",
  imprimeArrow(){
    setTimeout(() => {
      console.log(`resultado arrow: ${this.dato1} ${this.dato2}`);
    }, 1000)
  },
  imprimeNormal(){
    setTimeout(function() {
      console.log(`resultado normal: ${this.dato1} ${this.dato2}`);
    }, 1000)
  },
  pruebaArrow:()=>{
    console.log(`prueba arrow: ${this.dato1} ${this.dato2}`);
  },
  pruebaNormal(){
    console.log(`prueba normal: ${this.dato1} ${this.dato2}`);
  }
}

persona.imprimeArrow();
persona.imprimeNormal();
persona.pruebaArrow();
persona.pruebaNormal();
```

A diferencia del ejemplo anterior, dentro de su mismo contexto, una función flecha NO mantiene una referencia a ***this***

```
prueba arrow: undefined undefined
prueba normal: d1 d2
resultado arrow: d1 d2
resultado normal: undefined undefined
```

TypeScript

Object Destructuring

- Es una forma rápida de extraer información de objetos.

- Sintaxis:

let { <nombre de propiedades del objeto separados por coma> } = nombre del objeto

- Ejemplo:

```
let persona3 = {
  nombre3:"David",
  apellido3:"Gilmour",
  habilidad3:"Guitarra Electrica"
}
```

```
let n = persona3.nombre3;
let a = persona3.apellido3;
let h = persona3.habilidad3;

console.log(n,a,h);
```

Vs

```
let {nombre3,apellido3,habilidad3} = persona3;

console.log(nombre3,apellido3,habilidad3);
```

TypeScript

Object Destructuring

- Se puede definir alias para las variables:

- Ejemplo:

```
let {nombre3:n1,apellido3:a1,habilidad3:h1} = persona3;
console.log(n1,a1,h1);
```

TypeScript

Object Destructuring

- Con arreglos:

```
let pinkfloyd:string[] = ["The Wall", "The Darkside of the Moon","The Final Cut"];
let[album0,album1,album2] = pinkfloyd;
console.log(album0,album1,album2);
```

```
let pinkfloyd:string[] = ["The Wall", "The Darkside of the Moon","The Final Cut"];
// let[album0,album1,album2] = pinkfloyd;
let[,,album2] = pinkfloyd;
console.log(album2);
```

TypeScript

Promises

- Útiles para escribir código asíncrono.

- Facilita escribir código asíncrono en comparación con las funciones de callback.

- Toda la función asíncrona se puede referenciar por medio de un objeto, el cual puede ser utilizado como parámetro dentro de otra función.

```
let operacion = new Promise((resolve, reject){ //implementación })
let operacion2 = new Promise((resolve, reject){ //implementación }).then(operacion)
```

- Tienen 3 estados:
  - Pending: No ha sido ejecutada la función.
  - Resolved: La ejecución fue exitosa.
  - Rejected: Ocurrió un error en la ejecución o alguna de las condiciones no se cumplió.

TypeScript

Promises

Callback Vs. Promise

```
function algunaOperacion(function(err,  valor1) {
  operacion1(valor1, function(err, valor2){
    operacion2(valor2, function(err, valor3){
      operacion3(valor3, function(err, valor4){
        operacion4(valor4, function(valor5){
          //Algun procesamiento
        });
      });
    });
  });
});
```

```
//Cada operacionX es un Promise
algunaOperacion()
    .then(operacion1)
    .then(operacion2)
    .then(operacion3)
    .then(operacion4)
    .then(
        function(){
            //Se ejecuta cuando se llama a resolve();
        },
        function(){
            //Se ejecuta cuando se llama a reject();
        })
```

TypeScript

Promises

```
let promise = new Promise(function(resolve,reject){

  setTimeout(() => {
    console.log("Ejecucion asincrona finalizada...");
    //Si todo se eejecuto bien, invocamos reolve, de lo contrario reject.
    resolve();
    //reject();
  }, 3000)

})

console.log("Simulando alguna operacion 1");

promise.then(
  //Funcion que se invoca en resolve.
  function(){
    console.log("Ejecucion asincrona finzalizo con exito.");
  },
  //Funcion que se invoca en reject.
  function(){
    console.log("Ejecucion asincrona finzalizo con error.");
  }
)

console.log("Simulando alguna operacion 2");
console.log("Simulando alguna operacion 3");
```

TypeScript

Promises

También se puede utilizar con Funciones Flecha. Esto es útil si se quiere utilizar “this” dentro del Promise:

```
let promiseArrow = new Promise((resolve,reject) => {

  setTimeout(() => {
    console.log("Arrow: Ejecucion asincrona finalizada...");
    //Si todo se eejecuto bien, invocamos reolve, de lo contrario reject.
    resolve();
    //reject();
  }, 6000)

})
```

TypeScript

Clases y Constructores

- No permite el uso de múltiples constructores:

```
class Videojuego{
  nombre:string;
  developer:string;
  lanzamiento:number;

  constructor(){

  }

  constructor(nombre:string, developer:string, lanzamiento:number){
    this.nombre = nombre;
    this.developer = developer;
    this.lanzamiento = lanzamiento;
  }
}

let tlou:Videojuego = new Videojuego();
console.log(tlou);
let tlou2:Videojuego = new Videojuego("The Last of Us", "Naughty Dog", 2013);
console.log(tlou2);
```

TypeScript

Clases y Constructores

```
class Videojuego{
  nombre:string;
  developer:string;
  lanzamiento:number;

  constructor(nombre:string, developer:string, lanzamiento:number){
    this.nombre = nombre;
    this.developer = developer;
    this.lanzamiento = lanzamiento;
  }
}

let tlou2:Videojuego = new Videojuego("The Last of Us", "Naughty Dog", 2013);
console.log(tlou2);
```

```
▼ Videojuego {nombre: "The Last of Us", developer: "Naughty Dog", lanzamiento: 2013}
    developer: "Naughty Dog"
    lanzamiento: 2013
    nombre: "The Last of Us"
  ▶ __proto__: Object
```

TypeScript

Clases y Constructores

```
class Videojuego{
  nombre:string = "The Legend of Zelda: Breath of the Wild";
  developer:strin = "Nintendo";
  lanzamiento:num er = 2017;

  constructor(nombre:string, developer:string, lanzamiento:number){
    this.nombre = nombre;
    this.developer = developer;
    this.lanzamiento = lanzamiento;
  }
}

let tlou2:Videojuego = new Videojuego("The Last of Us", "Naughty Dog", 2013);
console.log(tlou2);
```

```
▼ Videojuego {nombre: "The Last of Us", developer: "Naughty Dog", lanzamiento: 2013}
    developer: "Naughty Dog"
    lanzamiento: 2013
    nombre: "The Last of Us"
  ▶ __proto__: Object
```

TypeScript

Decorators

- Son funciones, las cuales se establecen por medio del símbolo “@”.
- Los elementos que pueden ser *decorados* son: clases, parámetros, métodos, y propiedades.
- Representan metadatos para el elemento decorado, y son utilizados para configurar un comportamiento determinado de del elemento.

```
function Consola(target) {
  console.log('Clase decorada: ', target);
}

@Consola
class EjemploDecorador{
  propiedad:string;

  constructor(){

  }
}
```

```
Clase decorada:  ƒ EjemploDecorador() {
    }
```

TypeScript

Decorators

- Decoradores que vamos a estar utilizando:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  juego:string = "The Last of Us";
  plataforma:string = "Playstation 4";
  developer:string = "Naughty Dog";
  lanzamiento:number = 2013;
}
```

TypeScript

PRACTICA EN EQUIPO