

TypeScript

vs

JavaScript

JavaScript

Un poco de historia

- Nace en 1995 para añadir programas a la web.
- Implementado en el navegador **Nestcape**.
- Surgen las primeras aplicaciones web.
- Necesidad de ejecutar código en el navegador.
- Necesidad de modificar el DOM de forma dinámica.

Evolución de JavaScript

- **LiveScript, desarrollado por Brendan Eich (Nestcape)**
- **Nestcape y Sun Microsystems** apuestan por el lenguaje con el nombre de **JavaScript**.
- **Microsoft** incorporó su propia implementación **JScript**.
- JavaScript se hace popular y se estandariza con la especificación **ECMAScript**.
 - Desde entonces existen **11 versiones de ECMAScript**.

Contexto actual

- Cambio de paradigma. **JavaScript como centro de las tecnologías web.**
- Mucho más que para modificar el DOM.
- **JavaScript versátil:**
 - **Aplicaciones nativas híbridas** (Ionic + Capacitor o React Native)
 - **Desarrollo de aplicaciones de escritorio** (Angular PWA)
 - **SPAs** (Angular, React o Vue)
 - **Servidor** (NodeJS + Express)
 - **APIs** (Node + Express + Mongodb)
 - **Server Side Rendering** (Angular Universal o Next para React)
- Lenguaje predominante para **frontend y backend**.
- Frameworks y librerías

**¿Pero no es oro todo lo que
reluce?**

- **Weakly typed or untyped.**
- **Caótico, anárquico y propenso a errores.**
- **Lejos de la programación tradicional.**
- **JS inline**
- **Peticiones http innecesarias.**
- **Errores bloqueantes.**
- **Incompatibilidad entre navegadores.**

“Weakly typed” or “untyped” language



```
1 var myFirstNumber = 5;
2 var mySecondNumber = '10';
3
4 var operationResult = myFirstNumber + mySecondNumber;
5
6 console.log(operationResult);
7 // Output --> '510'
8
9 console.log(typeof(operationResult));
10 // Output --> string
11
12 var secondOperationResult = operationResult / 2;
13 console.log(secondOperationResult);
14 // Output --> 255
15
16 console.log(typeof(secondOperationResult));
17 // Output --> number
18
19 var thirdOperationResult = secondOperationResult + 'Hola mundo';
20 console.log(thirdOperationResult);
21 // Output --> 255Hola mundo
22
23 console.log(typeof(thirdOperationResult));
24 // Output --> string
```

**En ocasiones es caótico,
anárquico, indisciplinado y
propenso a errores.**



```
jQuery(document).ready(function($) {
    if ($.trim($(".CASUsername").html())!=='') {
        $("#menu-item-10161").addClass('hide');
        $("#menu-item-12252").removeClass('hide');
        $("#menu-item-13730").addClass('hide');
        $("#menu-item-13709").removeClass('hide');
    }

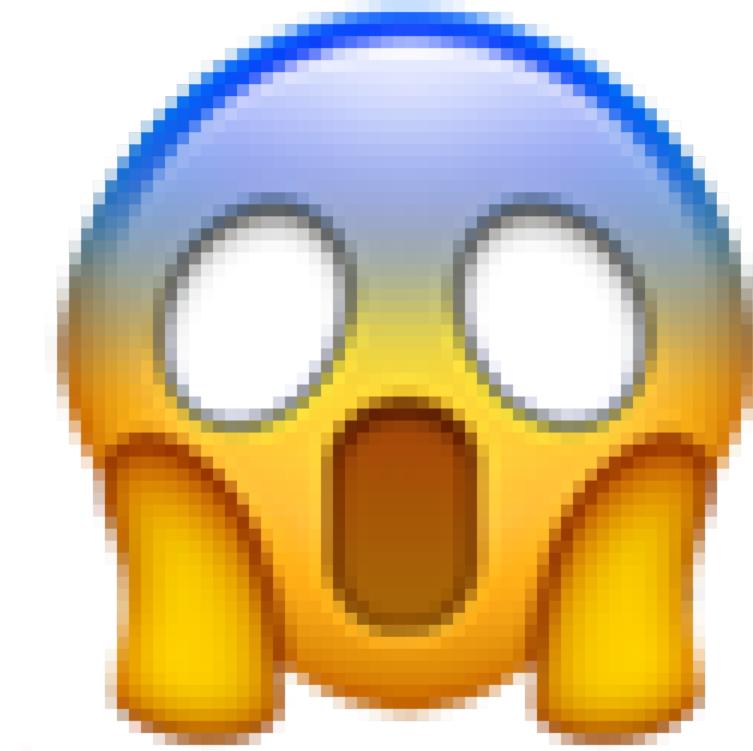
    try {
        var application = parameter.application;
        console.log(application);
        if(application === 'ekilibria'){
            var userLogueado = parameter.userloggedIn;
            var urlAreacliente = parameter.areaClienteUrl;
            var fromUrl = parameter.fromUrl;
            var locale = parameter.locale;

            locale = (locale !== 'es')? '/' + locale : ''; // si es idioma español no va el locale en la url
            var logout = 0;
            if(!CAS.isCasLoggedIn()){ // si no esta logueado en areacliente -> debo desloguearme de ekilibria

                if(fromUrl.indexOf('logout%3D1') > 0){ // si tiene el parametro logout en la url
                    logout = 1;
                    return '';
                }
                var positionFinal = fromUrl.indexOf('ekilibria-club-salud') + 20;
                var tamaño = fromUrl.length;
                var diferencia = tamaño - positionFinal;

                if(diferencia <= 3){ // para que no se haga un ciclo infinito, en la misma paginas
                    if(userLogueado !== ''){
                        location.href = urlEroski +locale + '/ekilibria-club-salud/?logout=1'; //redirigo a ekilibria
                    }
                }else{
                    location.href = urlEroski +locale +'/ekilibria-club-salud/?logout=1'; //redirigo a ekilibria y de
                }
            }else{ //esta logueado en areacliente
                var userEmail = CAS.getUserEmail();
                if(userLogueado !== userEmail){ // si el usuario que obtuve desde wordpress y el usuario que viene de
                    location.href='https://'+urlAreacliente+'/' + areacliente + locale + '/authentication?forceLogin=true&f
                }
            }
        }
    } catch(err) {
        console.log('no control');
    }
});
```

Javascript inline



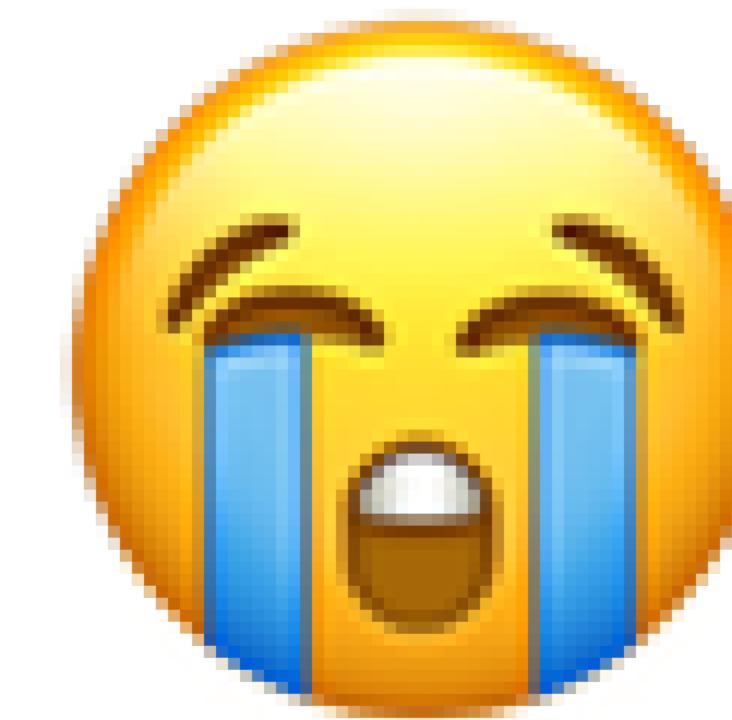
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    document.getElementById("container").innerHTML = "Common JS Bugs and Errors";
  </script>
  <div id="container"></div>
  <script src=".//caos.js"></script>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
```

✖ ▶ **Uncaught TypeError: Cannot set properties of null (setting 'innerHTML')**
at index.html:11

```
<script>
    document.getElementById("container").innerHTML = "Custom JS bugs and errors";
</script>
<div id="container"></div>
<script src="main.js"></script>
</body>
</html>
```

Peticiones HTTP innecesarias





Solución

- **Uso de frameworks JS** (Angular, React, Vue, Ionic...)
- **Uso de librerías de tipado** (TypeScript, JSX...)
- **Linters en el editor de código** (TypeScript, Prettier...)
- **Desarrollo de test unitarios** (Jasmine, Karma, Protractor...)
- **Librerías de UI** (Material Design, Bootstrap)
- **Herramientas de desarrollo para estándares modernos de JavaScript** (Webpack, gulp...)
- **Polyfills** para soportar incompatibilidades entre navegadores.

ECMAScript 6

¿Qué es ECMAScript?

- Especificación de por la que se rige JavaScript
- Estandarización.
- Implementación del lenguaje en los diferentes navegadores.
- Versiones
 - ES1, ES2, ES3, ES4, ES5 y ES6
 - ECMAScript 2016 / 2017 / 2018

Versiones E1 a E4

- **ECMAScript 1 · ES1 (1997)**
 - Primera especificación.
 - Características básicas de JavaScript
- **ECMAScript 2 · ES2 (1998)**
 - Actualizaciones editoriales
- **ECMAScript 3 · ES3 (1999)**
 - Expresiones regulares.
 - Try y catch
 - Switch
 - Do while
- **ECMAScript 4 · ES4**
 - Nunca llegó a salir a la luz.

ECMAScript 5 · ES5 (2009)

- Mayor revisión de la especificación
- **Características principales:**
 - **Strict mode:** Se habilita la directiva “use strict”. Pretende restringir algunas de las características iniciales de JavaScript.
 - **Acceso al carácter de un string a través del índice.**
 - **String multilinea.**
 - **Métodos de utilidad para arrays:**
 - Array.forEach()
 - Array.filter()
 - Array.map()
 - ...
 - **Gestión de JSON**
 - JSON.parse()
 - JSON.stringify():
 - **Manipulación de fechas**
 - **Getters y setters**
 - **Compatibilidad con navegadores**

ECMAScript 6 (ES6) 2015

- 2^a revisión que más características ha añadido
- Compatibilidad con navegadores.
 - **Babel:** <https://babeljs.io/>
 - **Polyfills:** <https://github.com/github/fetch>
- Existen versiones posteriores a ES6

Características principales

Declaración de variables con let

A partir de la revisión ES6 entra en juego una manera diferente de declarar las variables.

- Diferencia entre **var** y **let**
- Definición de variables para un ámbito

```
9  var a = 5;  
10 {  
11   let a = 10;  
12   console.log('Dentro del ámbito', a);  
13 }  
14 console.log('Fuera del ámbito', a);
```

Declaración de constantes con const

La palabra reservada `const` nos permite declarar valores constantes cuyo valor no se podrá cambiar a lo largo del código

```
20    // 2.1
21
22    const a = 5;
23    a = 10;
24
25    // 2.2
26
27    {
28        const b = 10;
29        console.log(b)
30    }
31    console.log(b);
```

Arrow functions

- Se mantiene la sintaxis de function tradicional
- Se añade una nueva sintaxis para simplificar la declaración =>.
 - No tienen su propio “this”
 - Deben estar definidas antes de su uso
 - Omitir la palabra reservada return

```
37 const foo1 = (arg1, arg2) => {
38   return arg1 + arg2
39 }
40
41 console.log(foo1(1,2));
42           (parameter) arg2: any
43 const foo2 = (arg1, arg2) => arg1 + arg2
44
45 console.log(foo2(1,2));
46
47 const foo3 = (arg1, arg2) => ({ value1: arg1, value2: arg2, sum: arg1 + arg2 })
48
49 console.log(foo3(1,2));
--
```

For / Of loop

Nueva característica para las estructuras de control repetitivas.

- Recorrer un array o un string sin necesidad de utilizar un contador.
- Variable que se le dará valor en cada iteración

```
55 const colors = ['Red', 'Blue', 'Green'];
56
57 for(let color of colors) {
58   | console.log(color);
59 }
60
61 const hello = 'Hola mundo';
62
63 for(let char of hello) {
64   | console.log(char);
65 }
```

Uso de clases

En ES6 se incluye el concepto de clases para la gestión de objetos en JavaScript.

```
71 class User {  
72   constructor(name, age) {  
73     this.name = name;  
74     this.age = age;  
75   }  
76 }  
77  
78 const usuario1 = new User('Igor', 32);  
79 console.log(usuario1);
```

Uso de promesas

- Gestión de tareas asíncronas.
- Una promesa debe ser resuelta, ya sea con un resultado satisfactorio o devolviendo un error

```
85 const multiplier = (number) => {
86   const promise = new Promise((resolve, reject) => {
87     const result: number = number;
88     if(result > 50) {
89       reject('El número es muy alto')
90     }
91
92     setTimeout(() => {
93       resolve(result);
94     }, 1000)
95   })
96
97   return promise
98 }
99
100 multiplier(6)
101   .then(res => {
102     console.log(res);
103     return multiplier(res);
104   })
```

Valores por defecto de los parámetros de una función

```
138  // OK
139  ↘ const foo = (arg1, arg2 = 2) => {
140  |   return arg1 * arg2;
141  }
142
143  console.log(foo(4));
144
145  // KO
146  ↘ const fooK0 = (arg2 = 2, arg1) => [
147  |   return arg1 * arg2;
148  ]
149
150  console.log(fooK0(4));
```

Parámetro rest de una función

```
156     function sum(...args) {  
157         let sum = 0;  
158         for (let arg of args) sum += arg;  
159         return sum;  
160     }  
161  
162     let x = sum(4, 9, 16, 25, 29, 100, 66, 77);  
163  
164     console.log(x);
```

Operador Spread

Permite modificar objetos de forma sencilla. Cambiar la propiedad de un objeto manteniendo las existentes es muy sencillo con este operador, pero también concatenar o clonar un array. Para hacer uso del operador utilizaremos la sintaxis de los puntos suspensivos por delante de un objeto o array.

```
172 let user = {  
173   name: 'Igor',  
174   age: 32,  
175 }  
176  
177 console.log(user);  
178  
179 let userModified = {  
180   ...user,  
181   age: 33,  
182   company: 'Worköhölics'  
183 }  
184  
185 console.log(userModified);
```

Métodos sobre Arrays o métodos de iteración

- Array.map()
- Array.filter()
- Array.reduce()
- Array.find()
- Array.findIndex()

```
220  var numbers1 = [45, 4, 9, 16, 25];
221  var numbers2 = numbers1.map(myFunction);
222
223  ↘ function myFunction(value) {
224    |   return value * 2;
225  }
226
227  console.log(numbers2);
228
229 // Arrow function de ES6
230
231  var numbers1 = [45, 4, 9, 16, 25];
232  var numbers2 = numbers1.map(n => n * 2);
233
234  console.log(numbers2);
```

P2 Práctica - ES6

P2 Práctica - ES6

Recursos de la práctica:

<https://github.com/igornieto/curso-angular-avanzado>

Objetivo:

- Descarga el archivo “p2-es6.zip”
- Completa el ejercicio siguiendo los comentarios de /index.js

TypeScript

¿Qué es TypeScript?

- Lenguaje de programación “**strongly typed**”
- **Open Source**, creado y mantenido por **Microsoft**
- Superconjunto de JavaScript
- **Nueva sintaxis** o sintaxis complementaria
- No añade nuevas características fuera de ECMAScript
- **Objetivo:**
 - **Estructuración optima** del código.
 - **Tipado de variables, objetos, métodos y argumentos.**
 - **Compatibilidad con navegadores** y diferentes especificaciones
 - **Control y detección temprana de errores.**
 - **Integración con editores de código.**

¿Qué es TypeScript?

- **Proyectos más sostenibles.**
- Adaptaciones de frameworks a TS.
 - Angular (lenguaje base)
 - React (versión soporte TS)
 - Vue (versión soporte TS)
- **TypeScript, es un lenguaje al alza (State of JS):**
 - 78% de los desarrolladores que usan JavaScript utilizan TypeScript
 - 93% de los desarrolladores que lo han utilizado lo volverían o volverán a utilizar

¿Cómo funciona TS?

TypeScript necesita ser transpilado para ser interpretado por el navegador. El código resultante es JavaScript y es totalmente compatible con cualquier entorno que interprete este lenguaje, por ejemplo, un navegador o un servidor basado en Node.js.

Este lenguaje además de estructurar nuestro código nos **permite el uso de estandares modernos de JS especificados a través de ECMAScript**. El código transpilado será compatible con los navegadores.

TypeScript no genera código JS adicional, es una herramienta que ayuda al programador en su entorno de desarrollo.

TypeScript playground

Aunque en un futuro desarrollaremos en TypeScript sobre Angular, **para aprender la sintaxis del lenguaje utilizaremos el playground de TypeScript** de cara a que lo probemos y consigamos plasmar los diferentes ejemplos que nos ayudarán a entenderlo.

<https://www.typescriptlang.org/play>

TypeScript Basics

Static Type Checking

TypeScript lleva a cabo “**static type checking**” con el fin de encontrar errores o verificar el código previo a su ejecución.

Como desarrolladores queremos evitar errores o bugs, pero ese trabajo no siempre es fácil con JavaScript.

Gracias a TS podremos detectar los errores y verificar el código tanto en el editor como en la consola a través del error del transpilador.

```
const message = "Hello World!";  
  
message()
```

Este error es evidente, pero en desarrollos JS complejos puede que tengamos que estirar mucho del hilo, depurar o forzar el error para conseguir solventar un bug. **TypeScript es una herramienta que nos permite controlar los errores y comprobar la estructura o tipos de elementos JS antes de que el programa sea ejecutado a través del static type checking.**

Non-exception Failures

Existen fallos en el desarrollo que no tienen porqué implicar un error JS.

Por ejemplo, en JS, una propiedad o variable tiene el valor “undefined” cuando no se ha definido. Al acceder a dicha variable o devolverla por consola, no implicará un error.

En cambio, al acceder a la variable con TypeScript el transpilador o el editor nos avisará de que ese valor no está definido.

Gracias a esta característica podremos controlar un posible error de forma más sencilla.

```
1 const user = {  
2   name: "Daniel",  
3   age: 26,  
4 };  
5  
6 user.location;
```

Herramientas de TS en el editor de código.

TypeScript nos facilita herramientas integradas en el IDE para **adelantarse al desarrollador con el fin de prevenir errores:**

- Sugerencias de acceso a las propiedades de variables u objetos.
- Sugerencias de acceso a los métodos disponibles en una variable u objeto.
- Ayudas rápidas de resolución de errores.
- Reorganización automática del código.
- Imports automáticos.

[https://www.typescriptlang.org/play?
importHelpers=true](https://www.typescriptlang.org/play?importHelpers=true)

El único requisito es que el editor de código que usemos sea compatible con TypeScript.

Tipado explícito

En general, en TS definiremos los tipos de forma explícita. Lo haremos a través de los “:” y las palabras reservadas para el tipado de parámetros, variables u objetos.

```
let username: string = 'Igor';
let age: number = 32;
```

 Ejemplo

Tipado implícito

Habrá ocasiones en las que el valor que le demos a una variable sea el que especifique el tipo de la variable. Este caso se denomina **inferencia**.

```
let username = 'Igor';
let age = 32;
```

```
username = 489;|
```

Transpilador TypeScript

TypeScript necesita ser transpilado a JS para que sea interpretado por el navegador, para ello tendremos que utilizar un transpilador.

Normalmente el compilador irá **integrado en el framework que estemos utilizando, en este caso Angular**. Aun así también podemos compilar el TS directamente en la consola.

Instalación para transpilación manual en consola:

- Instalar package TypeScript de forma global a través de NPM.
- Transpilar archivo .ts (extensión de TS a través del comando tsc)

Downleveling

Como hemos indicado previamente, una de las tareas del transpilador de TS es hacer que el código sea **interpretable por los navegadores**.

TS tiene la **capacidad de adaptarse a las diferentes especificaciones de ECMAScript**. Para ello podremos indicar un --target con el fin de indicar al compilador que la tiene que ser soportada por una versión específica.

Borrado de tipos

Cuando transpilamos el código TS, la definición de tipos que se han declarado se borrarán, ya que no son interpretables y es código irrelevante para el navegador.

Tipos de TypeScript

string, number & boolean

Los tipos más comunes de TS vienen definidos en base a los **7 tipos primitivos de JavaScript**.

- **string:** Cadenas de caracteres
- **number:** Tanto para enteros como decimales.
- **boolean:** true or false

A continuación veremos un ejemplo de tipado de los 3 casos.

```
const user: string = 'Ramón';
const user2 = 'Alex';
const age: number = 35;
const age2 = 40;
const price: number = 102.5;
const price2 = 90.2;
const isActive: boolean = true;
const isHover = false;

console.log(typeof user);          // string
console.log(typeof user2);         // string
console.log(typeof age);           // number
console.log(typeof age2);          // number
console.log(typeof price);         // number
console.log(typeof price2);        // number
console.log(typeof isActive);      // boolean
console.log(typeof isHover);       // boolean
```

Arrays

Para especificar el tipo de un array podemos hacerlo con los **corchetes precedido del tipo de elemento del que está compuesto el array** (`number[]`, `string[]...`).

Otra sintaxis de declaración de arrays es **a través de la palabra reservada `Array<T>`** (`Array<number>`, `Array<string>...`).

```
const arrayOfNumbers: number[] = [1,2,3,4,5];
const arrayOfNumbers2: Array<number> = [1,2,3,4,5];

const arrayOfStrings: string[] = ['Blue', 'Green', 'Red'];
const arrayOfStrings2: Array<string> = ['Blue', 'Green', 'Red'];
```

Any

Tipo que anula el tipado de un elemento y lo habilita para que pueda tener cualquier tipo de valor.

Se recomienda su uso solo cuando sea estrictamente necesario.

Tomarse un tiempo para tipar los elementos de nuestro proyecto será muy positivo de cara a no acumular deuda técnica y hacer un proyecto más sostenible.

Usar any por defecto es ir en contra de los beneficios de TS

```
let obj: any = { x: 0 };
// None of the following lines of code will throw compiler errors.
// Using 'any' disables all further type checking, and it is assumed
// you know the environment better than TypeScript.
obj.foo();
obj();
obj.bar = 100;
obj = "hello";
const n: number = obj;
```

Any implícito

Por defecto, **cuando no tipamos un valor, TS lo interpreta como any** a no ser que se le de un valor por defecto que a través de la inferencia le de un tipo específico.

Normalmente esto es algo que queremos evitar ya que el “**type-checker**” no podrá interpretar el tipo.

Si queremos que el any implícito se interprete como un error podemos hacerlo en la compilación manuala traves del siguiente flag:

--noImplicitAny

O editando la configuración en el tsconfig.json

Especificación de tipo en variables

Cuando utilizamos una variable a través de var, let o const podemos especificar su tipo de forma explícita de la siguiente manera.

Como he explicado en la sección anterior habrá ocasiones en que el tipo se especificará automáticamente a través de la inferencia al asignarle un valor a la variable.

```
let myName: string = "Alice";  
let myName2 = "Alice";|
```

Especificación de tipo en funciones

TypeScript nos da la posibilidad de especificar los tipos de los diferentes inputs y outputs de una función:

- Parámetros de una función
- Return

```
function saludo(nombre: string) : string {  
    return `Hola ${nombre}`;  
}  
  
console.log(saludo('Alex'));  
  
const multiplicador = (n1: number, n2: number): number => {  
    return n1 * n2;  
}  
  
console.log(multiplicador(5, 6));
```

Especificación de tipo en objetos

Los especificación de tipos más comunes después de las primitivas son sobre los objetos. Este tipo de especificación **nos permite dar un tipo concreto a cada una de sus propiedades de un objeto.**

Propiedades opcionales

En un objeto podremos tener propiedades cuyo **valor no tenga que ser requerido**, para ello contamos con las propiedades opcionales para las que utilizaremos la “?” por detrás de la propiedad. En este caso al definir el valor del objeto podremos dejar sin definir las propiedades que definamos de dicha manera.

```
const user : { name: string, age: number, isAdmin: boolean } = {  
    name: 'Igor',  
    age: 32,  
    isAdmin: true  
}  
  
const user2 : {  
    name: string,  
    age: number,  
    isAdmin: boolean,  
    company?: string  
} = {  
    name: 'Igor',  
    age: 32,  
    isAdmin: true  
}
```

Union Types

Característica para dar **más de un tipo a una variable, argumento o propiedad.**

Por ejemplo, podemos definir que a una variable se le puede asignar tanto un número como un string, o utilizarlo para especificar los posibles valores de los parámetros de una función.

Esta característica suele ser muy **útil a la hora de declarar argumentos de una función que pueden ser de diferentes tipos.**

```
function printId(id: number | string) {  
  console.log("Your ID is: " + id);  
}  
// OK  
printId(101);  
// OK  
printId("202");
```

Alias Type

Esta característica permite **asignar un alias a un tipo o listado de propiedades**.

En vez de definir los tipos directamente sobre las variables, propiedades o argumentos, puede haber casos en los que veamos la necesidad de **reutilizar** un grupo de propiedades, una unión type o simplemente contextualizar un tipo a través de un alias.

```
type Username = string;
type Id = string | number;

type User = {
  id: Id;
  name: Username;
  age: number;
  isAdmin: boolean;
}

const user1: User = {
  id: '155e77ee-ba6d-486f-95ce-0e0c0fb4b919',
  name: 'Alex',
  age: 32,
  isAdmin: true
}

const user2: User = {
  id: 14,
  name: 'Xabi',
  age: 50,
  isAdmin: false
}

console.log(user1);
console.log(user2);
```

 Ejemplo

Interface

Otra forma de **reutilizar un grupo de tipos es a través de una interfaz**. De esta manera podremos **modelar las propiedades de un tipo de objeto** y utilizarlo en diferentes puntos del proyecto. Al fin y al cabo tanto “**type**” como “**interface**” son formas para la reutilización de tipado.

```
type Username = string;
type Id = string | number;

interface User {
  id: Id;
  name: Username;
  age: number;
  isAdmin: boolean;
}

const user1: User = {
  id: '155e77ee-ba6d-486f-95ce-0e0c0fb4b919',
  name: 'Alex',
  age: 32,
  isAdmin: true
}
const user2: User = {
  id: 14,
  name: 'Xabi',
  age: 50,
  isAdmin: false
}

console.log(user1);
console.log(user2);
```

Ejemplo

Las diferencias entre una “interface” y un “type” son las siguientes:

- La **sintaxis para extender una “interface” es mucho más amigable** que la de extender un “type”.
- En una “interface”, a diferencia del “type” puedes **añadir nuevas propiedades redefiniendo la “interface” las veces que sea necesario**.

```
type Username = string;
type Id = string | number;
type Role = 'developer' | 'designer' | 'marketing';

interface User {
  id: Id;
  name: Username;
  age: number;
  isAdmin: boolean;
}

interface Employee extends User {
  role: Role;
}

const user: Employee = {
  id: 'dc523cb313b63dfe5be2140b0c05b3bc',
  name: 'Marta',
  age: 40,
  isAdmin: false,
  role: 'developer'
}
```

Ejemplo

```
type Username = string;
type Id = string | number;
type Role = 'developer' | 'designer' | 'marketing';

type User = {
  id: Id;
  name: Username;
  age: number;
  isAdmin: boolean;
}

type Employee = User & {
  role: Role;
}

const user: Employee = {
  id: 'dc523cb313b63dfe5be2140b0c05b3bc',
  name: 'Marta',
  age: 40,
  isAdmin: false,
  role: 'developer'
}
```

Ejemplo

```
type Username = string;
type Id = string | number;
type Role = 'developer' | 'designer' | 'marketing';

interface User {
  id: Id;
  name: Username;
  age: number;
  isAdmin: boolean;
}

interface User {
  role: Role;
  location?: string;
}

const user: User = {
  id: 'dc523cb313b63dfe5be2140b0c05b3bc',
  name: 'Marta',
  age: 40,
  isAdmin: false,
  role: 'developer'
}
```

Ejemplo

```
type Username = string;
type Id = string | number;
type Role = 'developer' | 'designer' | 'marketing';

type User = {
  id: Id;
  name: Username;
  age: number;
  isAdmin: boolean;
}

type User = {
  role: Role;
  location?: string;
}

const user: User = {
  id: 'dc523cb313b63dfe5be2140b0c05b3bc',
  name: 'Marta',
  age: 40,
  isAdmin: false,
  role: 'developer'
}
```

Ejemplo

Aserciones de tipo

En ocasiones tenemos información de la que TS no es consciente. TS no siempre sabrá qué propiedades o métodos tiene disponibles.

Si nosotros tenemos más información que TS y sabemos la estructura de un elemento, podemos tiparlo a través del “as”.

Por ejemplo, un método u propiedad de manipulación del DOM o la respuesta de una API.

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
const context = myCanvas.getContext('2d');

/////

interface User {
  gender: string;
  name: {};
    title: string;
    first: string;
    last: string;
}
}

fetch('https://randomuser.me/api/')
  .then(res => res.json())
  .then(res => {
    //console.log(res)
    const user = res.results[0] as User;
    console.log(user.name);
  })
}
```

Enums

Las enumeraciones permiten al desarrollador **declarar posibles valores en un listado y así poder hacer referencia a un valor entre sus posibles valores de forma sencilla.**

Las enumeraciones podrán definirse a través de un listado simple o a través de un listado tipo clave-valor.

```
enum Shape {  
  Square,  
  Circle,  
  Triangle,  
}  
  
interface Composition {  
  name: string;  
  shape1: Shape;  
  shape2: Shape  
}  
  
const composition1: Composition = {  
  name: 'Composition 1',  
  shape1: Shape.Circle,  
  shape2: Shape.Square  
}  
  
enum Direction {  
  North = 'n',  
  South = 's',  
  East = 'e',  
  West = 'w'  
}  
  
console.log(Direction.East);
```

 Ejemplo

null & undefined

Existen 2 tipos primitivos en JavaScript para hacer referencia a la ausencia de valor o un valor no inicializado. TS sigue la misma línea y mantiene estos dos tipos con la misma sintaxis.

TS nos permitire configurar nuestro entorno para que los valores null y undefined se interprete de diferente manera:

- **strictNullChecks off**: Podemos acceder y utilizar los valores null y undefined como si fuesen cualquier otro valor sin que sean interpretados como un error por el TC.
- **strictNullChecks on**: No podemos utilizar ni acceder a valores null o undefined. Tendremos que comprobar programáticamente que un valor no sea null o undefined.

En el último caso el sufijo “!” nos permite sobre una variable o propiedad nos permite ignorar la comprobación. Es adecuado cuando sabemos que el valor en ningún caso será null o undefined aunque para TS pueda serlo

```
function doSomething(x: string | null) {  
  if (x === null) {  
    // do nothing  
  } else {  
    console.log("Hello, " + x.toUpperCase());  
  }  
  
}  
  
function liveDangerously(x?: number | null) {  
  // No error  
  console.log(x!.toFixed());  
}
```

 Ejemplo

Tipos menos comunes bigint y Symbol

En el primer punto hemos visto que **además de las 3 primitivas más comunes, null y undefined existían otros 2 tipos primitivos:**

- **bigint**: En ES2020 se introduce en JS un nuevo tipo primitivo para enteros con un valor muy alto.
- **symbol**: Esta es una primitiva de JS que permite declarar una referencia única y global a través de la función `Symbol()`

```
// Creating a bigint via the BigInt function
const oneHundred: bigint = BigInt(100);

// Creating a BigInt via the literal syntax
const anotherHundred: bigint = 100n;

console.log(anotherHundred);
```

Clases

TypeScript da **soporte total a las clases de JavaScript** introducidas en la especificación ES6 y aporta nuevas características que son compiladas con el fin de ser interpretadas por todos los navegadores.

A través de las clases de TypeScript podemos llevar a cabo programación orientada a objetos cercana a lenguajes de programación como Java o C#.

Declaración de una clase

La forma más sencilla de utilizar una clase es definir una clase vacía. Lo haremos a través de la palabra reservada `class` y le daremos el nombre que queramos.

En este caso una clase vacía no tiene ninguna utilidad, por lo que definiremos sus propiedades con el fin de crear un objeto útil.

```
class Point {  
    x: number;  
    y: number;  
}
```

```
const pt = new Point();  
pt.x = 0;  
pt.y = 0;
```

```
class Point {  
    x?: number;  
    y?: number;  
}
```

```
const pt = new Point();  
pt.x = 0;  
pt.y = 0;
```

Constructor

Si definimos la clase tal cual tendremos errores ya que las propiedades no tienen un valor por defecto y hemos establecido que su valor es requerido.

Por lo tanto o añadimos un valor por defecto a cada propiedad o las añadimos como opcionales o les damos un valor en el constructor.

```
class Point {  
    x: number;  
    y: number;  
}
```

```
class Point {  
    x: number = 0;  
    y: number = 0;  
}
```

```
class Point {  
    x: number;  
    y: number;  
    scale: number = .5;  
  
    constructor(x: number, y: number, scale?: number) {  
        if (scale) this.scale = scale;  
        this.x = x * this.scale;  
        this.y = y * this.scale;  
    }  
}
```

 Ejemplo

Declaración de métodos

En una clase podremos definir diferentes métodos para realizar diferentes operaciones o acciones sobre el objeto creado.

Para acceder a las propiedades dentro de un método utilizaremos el `this`, si lo hacemos sin el `this` estaremos haciendo referencia a un elemento definido en el ámbito del método.

```
class Point {  
    x: number;  
    y: number;  
    scale: number = .5;  
  
    constructor(x: number, y: number, scale?: number) {  
        if (scale) this.scale = scale;  
        this.x = x * this.scale;  
        this.y = y * this.scale;  
    }  
  
    fromOrigin() {  
        return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));  
    }  
}  
  
const point1 = new Point(324, 176);  
console.log(point1.fromOrigin());
```

Ejemplo

Getters y Setters

Con el fin de acceder o dar valor a las diferentes propiedades de un objeto podremos definir métodos a través de las palabras reservadas get y set.

El get siempre devolverá un valor, el set no devolverá valor alguno, solo realizará una actualización del valor de una propiedad.

```
class Point {  
    x: number;  
    y: number;  
    scale: number = .5;  
  
    constructor(x: number, y: number, scale?: number) {  
        if (scale) this.scale = scale;  
        this.x = x * this.scale;  
        this.y = y * this.scale;  
    }  
  
    fromOrigin() {  
        return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));  
    }  
  
    get getX() {  
        return this.x;  
    }  
  
    set setX(value: number) {  
        this.x = value * this.scale;  
    }  
  
}  
  
const point1 = new Point(324, 176);  
console.log(point1.fromOrigin());  
console.log(point1.getX());  
point1.setX = 500;  
console.log(point1.getX());
```

Ejemplo

Herencia de clases

Extensión de clases

Con objetivo de reutilizar código podremos extender clases.

Usaremos la palabra reservada extends tras la definición del nombre de la clase seguida de la clase que queremos extender. Para acceder a las propiedades de la clase padre lo haremos a través del super().

```
class Animal {  
    move() {  
        console.log("Moving along!");  
    }  
}  
  
class Dog extends Animal {  
    woof(times: number) {  
        for (let i = 0; i < times; i++) {  
            console.log("woof!");  
        }  
    }  
}  
  
const d = new Dog();  
// Base class method  
d.move();  
// Derived class method  
d.woof(3);
```

Implementar clases

Como en cualquier otro lenguaje orientado a objetos podremos utilizar la herencia implementando una clase. En este caso, a diferencia de con extends, deberemos sobreescibir los métodos del padre en la clase hija para que sea una clase válida.

```
interface Movable {
  move(): void;
}

class Dog implements Movable {

  move() {
    console.log("I'm moving");
  }

  woof(times: number) {
    for (let i = 0; i < times; i++) {
      console.log("woof!");
    }
  }
}

const d = new Dog();
// Base class method
d.move();
// Derived class method
d.woof(3);
```

 Ejemplo

Módulos de TS

A lo largo de la historia de JavaScript se ha tratado de modularizar el código de diferentes formas, desde 2012 TS ha implementado soporte para diferentes formatos **hasta que llegó la especificación ES6 en el año 2015.**

En ese momento se estandarizó la forma en modular el código tanto en JS como en TS haciendo uso de los ES Modules (ECMAScript Modules).

Para definir un módulo utilizaremos las **palabras reservadas import / export**. Todo código JS que no utilice el import / export no se considerará módulo si no script.

```
export default <element>  
import <name> from <path>
```

Si queremos **exportar un módulo como principal o “main module” utilizaremos “default”**. En ese caso, importaremos el módulo a través de un nombre y el path específico de donde leera el módulo default.

Solo podrá haber un módulo principal por archivo.

```
class Animal {  
  move() {  
    console.log("Moving along!");  
  }  
}  
  
class Dog extends Animal {  
  woof(times: number) {  
    for (let i = 0; i < times; i++) {  
      console.log("woof!");  
    }  
  }  
}  
  
export default Dog  
  
import Dog from "./module1";  
const dog1 = new Dog();
```

```
export <element>
import {<name1>, <name2>...} from <path>
```

En el caso de que queramos exportar más de un elemento desde un mismo módulo deberemos omitir la palabra default y exportar tantas variables, constantes, métodos... como queramos.

```
export class Animal {  
    move() {  
        console.log("Moving along!");  
    }  
}  
  
export class Dog extends Animal {  
    woof(times: number) {  
        for (let i = 0; i < times; i++) {  
            console.log("woof!");  
        }  
    }  
}
```

```
import {Dog, Animal} from './module2';  
const dog = new Dog();  
const animal = new Animal();
```

En el último caso **si no queremos ir importando los elementos de un en uno** a uno podemos utilizar el operador * en el import y asociar todos los elementos a un namespace.

```
import * as Zoo from './module2';
const dog = new Zoo.Dog();
const animal = new Zoo.Animal();
```

P3 Práctica – TypeScript

Recursos de la práctica:

<https://github.com/igornieto/curso-angular-avanzado>

Objetivo:

- Descarga el archivo “p3-typescript.zip”
- Completa el ejercicio siguiendo los comentarios de /index.js

En este punto podríamos profundizar mucho más en TS a través de ejemplos, diferentes casuísticas y características concretas, pero lo dejaremos aquí.

Aprender TS no es el objetivo principal del curso, aunque me parece muy importante conocer sus características principales tal y como hemos hecho para que tengamos **soltura con el lenguaje de programación en el que se basa Angular**.