



Desarrollo y Programación de
Aplicaciones Avanzadas con Angular

Formación online

OBJETIVOS

Módulo 1. Intro a TypeScript

Módulo 3. AppModule & AppComponent

Módulo 5. Data Binding

Módulo 7. Services / HTTP Req / RxJS

Módulo 9. Pruebas unitarias

EVALUACIÓN FINAL

PROGRAMACIÓN DIDÁCTICA

Módulo 2. Introducción a Angular

Módulo 4. Comunicación entre componentes

Módulo 6. Angular Directives

Módulo 8. Angular Routing

Módulo 10. Deploy



MIGUEL CAMPOS RIVERA

DOCENTE

Si tienes cualquier consulta o necesitas contactar con
el/la docente, escríbeme a
camposmiguel@gmail.com

Objetivos

La presente formación persigue, que el alumnado de **Banco Santander**, al finalizar la formación sea capaz de:

- Aprender desarrollar aplicaciones web con Angular que se conecten a fuentes de datos (API Rest) mediante servicios HTTP.
- Aprender los components y directivas principales de Angular.
- Diseñar formularios reactivos.
- Aprender sobre el DI, services, components.
- Hacer un deploy de una app Angular.
- Mejorar la versatilidad en el puesto de trabajo, por la aplicación de los conocimientos y técnicas adquiridas en este curso.



CHECK-IN

Presentación

- Nombre
- Trayectoria (formación, tecnologías en las que estoy especializado)
- Conocimiento sobre Angular
- ¿Qué espero del curso?

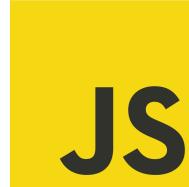


Módulo 1. Intro a TypeScript

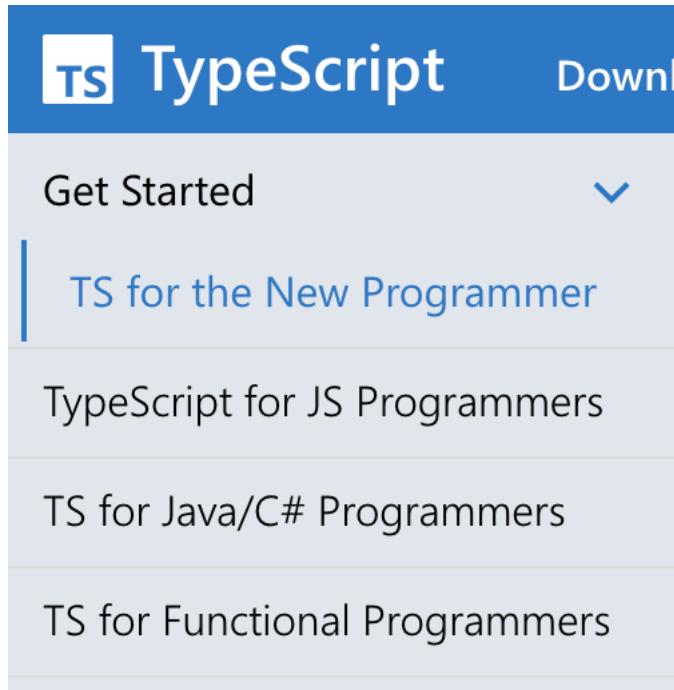


Módulo 1. Intro a TypeScript

¿Qué es  TypeScript ?

¿Qué relación tiene con  JavaScript ?

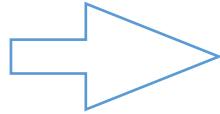
Módulo 1. Intro a TypeScript



<https://www.typescriptlang.org>

Módulo 1. Intro a TypeScript

Para responder bien a la pregunta, hagamos un poco de historia...



- JavaScript (conocido también como **ECMAScript**) nace como un lenguaje para navegadores web. Fue inventado para embeber pequeños bloques de código en el lado del cliente.
- Apenas se escribían 12 líneas de código.
- Se hizo popular y los web developers comenzaron a hacer aplicaciones más extensas, de cientos de LOC.

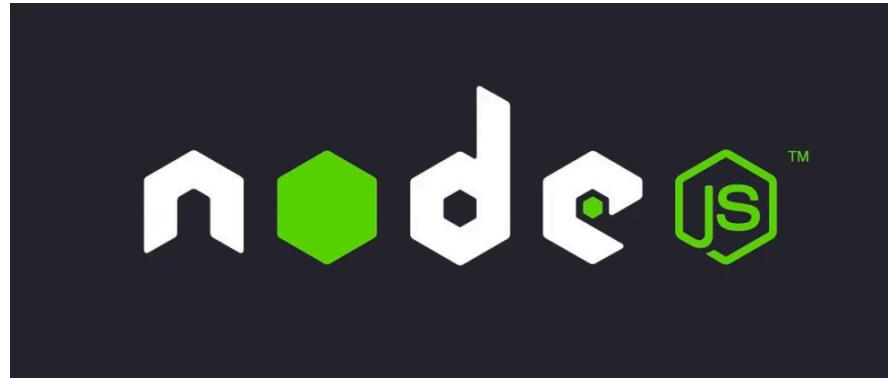


+



Web Browser

Módulo 1. Intro a TypeScript



Pero lo que ha vuelto a hacer popular a JavaScript ha sido el desarrollo de aplicaciones en el lado del servidor con Node.js

The “run anywhere” nature of JS makes it an attractive choice for cross-platform development.

Módulo 1. Intro a TypeScript



The image shows a screenshot of the V8 project website. The header features a blue navigation bar with the V8 logo on the left and six menu items: Home, Blog, Docs, Tools, JS/Wasm features, and Research. The 'Blog' item is highlighted with a blue background. Below the header, the main content area has a white background and displays the title 'What is V8?' in large, bold, black font. A detailed description follows, explaining that V8 is Google's open source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and Node.js, implements ECMAScript and WebAssembly, and runs on various operating systems and processor architectures. It can run standalone or be embedded into C++ applications.

What is V8?

V8 is Google’s open source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and in Node.js, among others. It implements [ECMAScript](#) and [WebAssembly](#), and runs on Windows 7 or later, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors. V8 can run standalone, or can be embedded into any C++ application.

<https://v8.dev/>

Módulo 1. Intro a TypeScript

JS

- Pero JavaScript, como cualquier otro lenguaje, presenta algunas rarezas o sorpresas como las siguientes:

```
if ("" == 0) {  
    console.log(" entra en el if");  
}
```

```
var x = 4;  
if (1 < x < 4) {  
    console.log(" entra en el if para cualquier valor de x");  
}
```

```
const obj = { width: 10, height: 15 };  
const area = obj.width * obj.height;  
console.log(area);
```

Módulo 1. Intro a TypeScript



TypeScript

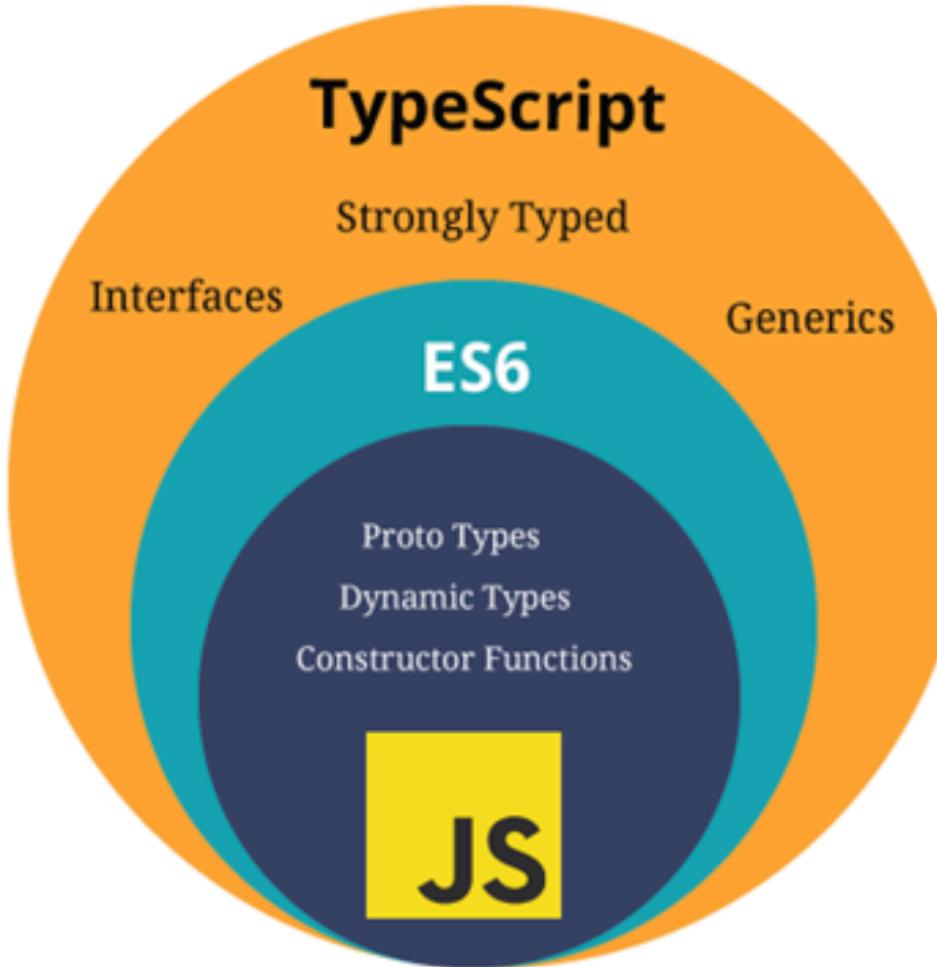


Microsoft

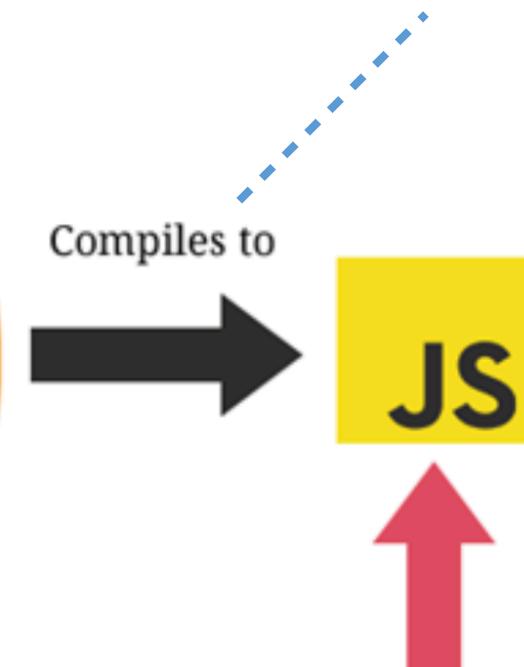
A Typed Superset of JavaScript

“TypeScript is a language that is a superset of JavaScript: JS syntax is therefore legal TS”

Módulo 1. Intro a TypeScript



Al proceso de traducción de TypeScript
A JavaScript, se le llama “transpilación”



The Browser
can't run TypeScript

Módulo 1. Intro a TypeScript

Scripting engine conformance

Scripting engine	Reference application(s)	Conformance ^[16]			
		ES5 ^[17]	ES6 (2015) ^[18]	ES7 (2016) ^[19]	Newer (2017+) ^{[19][20]}
SpiderMonkey	Firefox 94	100%	98%	100%	100%
V8	Google Chrome 95, Microsoft Edge 95, Opera 80	100%	98%	100%	100%
JavaScriptCore	Safari 15	100%	99%	100%	90%

https://en.wikipedia.org/wiki/ECMAScript#12th_Edition_%E2%80%93_ECMAScript_2021

Módulo 1. Intro a TypeScript



- Syntax

```
let a = (4  
      ^)  
')' expected.
```

- Typed

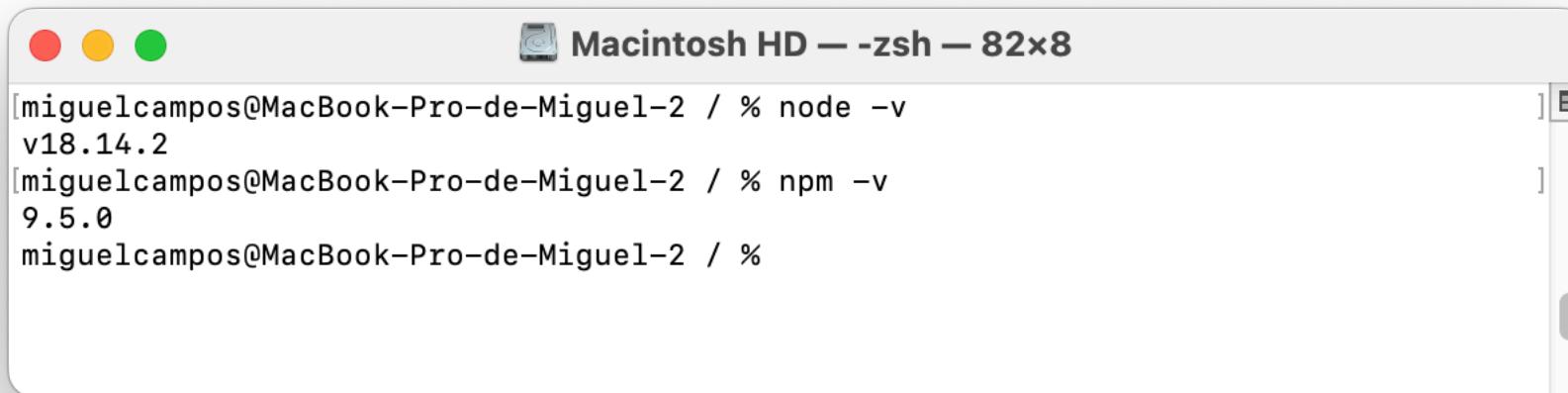
```
console.log(4 / []);
```

The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.

Módulo 1. Intro a TypeScript

Preparando el entorno

Instalamos Nodejs y npm: <https://nodejs.org/es/>



A screenshot of a macOS terminal window titled "Macintosh HD — zsh — 82x8". The window shows the following command-line session:

```
miguelcampos@MacBook-Pro-de-Miguel-2 ~ % node -v
v18.14.2
miguelcampos@MacBook-Pro-de-Miguel-2 ~ % npm -v
9.5.0
miguelcampos@MacBook-Pro-de-Miguel-2 ~ %
```

Módulo 1. Intro a TypeScript

Preparando el entorno

Instalamos TypeScript

<https://www.npmjs.com/package/typescript>



TypeScript

CI CI passing | Azure Pipelines Azure Pipelines never built | npm package 4.3.5 | downloads 89M/month

Install

npm i typescript

A screenshot of a macOS terminal window titled 'miguelcampos -- zsh -- 82x5'. The window shows the command 'tsc -v' being run, with the output 'Version 4.9.5' displayed.

DECLARACIÓN DE VARIABLES

Módulo 1. Intro a TypeScript

- Utilizamos la siguiente sintaxis:

```
variableName: type
```

- Declaración de variable con tipo:

```
let x: number;
```

- Declaración de variable con inferencia de tipo (similar a JavaScript):

```
let y = 1; // le estamos infiriendo el tipo number
```

- Declaración de variable sin inicialización:

```
let z;
```

Módulo 1. Intro a TypeScript

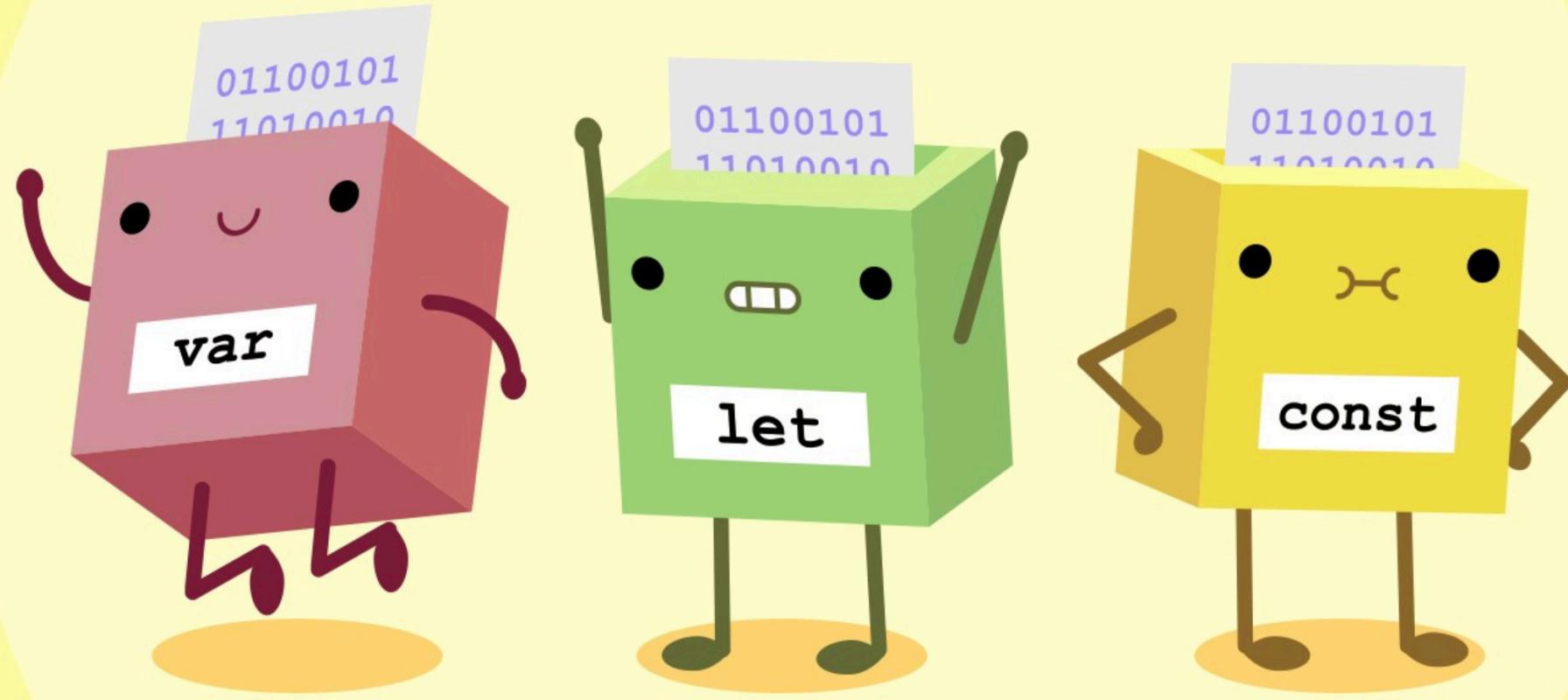
- En el ejemplo anterior, TypeScript infiere el tipo **number** a la variable **y**
- ¿Qué ocurre si intenta asignar un tipo de valor diferente?

Escribimos `y = "one"`. Tal como se esperaba, esto genera el error **El tipo "string" no se puede asignar al tipo "number"** porque la comprobación de tipos estáticos no permite que un elemento `string` se asigne a la variable.

- ¿Y qué ocurre con la variable **z** que no tiene tipo ni valor asignado?

(TypeScript ha inferido que `z` es de tipo `any` porque no ha asignado un tipo o lo ha inicializado cuando se ha declarado. Obtendrá más información sobre el tipo `any` más adelante).

Módulo 1. Intro a TypeScript



Módulo 1. Intro a TypeScript

- En **ES6** tenemos tres tipos de variables: var, let y const.
 - **var** se utiliza para declarar una variable y adicionalmente se puede inicializar el valor de esta variable. Por ejemplo: **var i = 0;**
 - **Ejemplo:** Si declaramos la variable dentro de una función, el valor de nuestra variable vivirá solamente dentro del scope de esta función.

```
function explainVar(){
  var a = 10;
  console.log(a); // output 10

  if(true){
    var a = 20;
    console.log(a); // output 20
  }

  console.log(a); // output 20
}
```

Módulo 1. Intro a TypeScript

- **Ejemplo 2:** Si olvidas declarar una variable dentro de tu función o *loop* (en este caso no declaramos *i* dentro del *for*), el intérprete de Javascript la va a declarar de forma global. El valor de *i* a nivel global será reasignado por el *for loop*.

```
var i = 60;
(function explainVar(){
  for( i = 0; i < 5; i++){
    console.log(i) //Output 0, 1, 2, 3, 4
  }
})();
console.log("Despues del loop", i) // Output 5
```

Módulo 1. Intro a TypeScript

- **Let**
- Ejemplo: funciona de acuerdo a lo esperado. Esto pasa porque *let* introduce el “**block scope**”. La variable asignada como *let* solo será accesible dentro del *for loop*.

```
var i = 60;
(function explainVar(){
  for(let i = 0; i < 5; i++){
    console.log(i)                  //Output 0, 1, 2, 3, 4
  }
})();
console.log("Despues del loop", i) // Output 60
```

```
function explainLet(){
  let a = 10;
  console.log(a); // output 10
  if(true){
    let a = 20;
    console.log(a); // output 20
  }
  console.log(a); // output 10
}
```

Módulo 1. Intro a TypeScript

- **Const**: es igual que *let*, con una pequeña gran diferencia: no puedes re asignar su valor.
- Ejemplo: La consola mostrara un error cuando tratemos de re asignar el valor de una variable *const*

```
function explainConst(){
  const x = 10;
  console.log(x); // output 10
  x = 20; //throws type error
  console.log(x);
}
```

Módulo 1. Intro a TypeScript

TIPOS EN TYPESCRIPT

cualquiera (any)

Tipos primitivos

booleano
número
cadena
enumeración
no válido

Tipos de objeto

clase
interfaz
matriz
literales

Tipo parámetros

NULL
no definido

Módulo 1. Intro a TypeScript

TIPOS BÁSICOS

Boolean

- El tipo de datos más básico es el valor `true` o `false`, conocido como booleano.
- Ejemplo:

```
let flag: boolean;  
let yes = true;  
let no = false;
```

Módulo 1. Intro a TypeScript

Number

- Al igual que en JavaScript, todos los números en TypeScript son valores de número de punto flotante o BigIntegers. Estos números de punto flotante obtienen el tipo `number`, mientras que los valores BigIntegers obtiene el tipo `bigint`
- Ejemplo:

```
let x: number;  
let y = 0;  
let z: number = 123.456;  
let big: bigint = 100n;
```

Módulo 1. Intro a TypeScript

String

- La palabra clave `string` representa secuencias de caracteres almacenados como unidades de código Unicode UTF-16. Al igual que JavaScript, TypeScript también usa comillas dobles ("") o comillas simples ('') para rodear los datos de cadena.
- Ejemplo:

```
let s: string;  
let empty = "";  
let abc = 'abc';
```

String

- En TypeScript, también puede usar cadenas de plantilla (Template literal), que pueden abarcar varias líneas y tener expresiones insertadas. Estas cadenas están rodeadas por el carácter de comilla simple/tilde grave (`) y las expresiones insertadas tienen el formato \${ expr }.
- Ejemplo:

```
let name: string = `Miguel Campos`;
let sentence: string = `Hola, mi nombre es ${name}.
y pronto tendré ${age + 1} años.`;
```

Módulo 1. Intro a TypeScript

void, null y undefined

- JavaScript y TypeScript tienen dos valores primitivos que se usan para indicar un valor ausente o con inicialización anulada: `null` y `undefined`. Estos tipos son más útiles en el contexto de las funciones, por lo que los trataremos con las funciones.

Módulo 1. Intro a TypeScript

Enumeraciones (enum)

- Las enumeraciones permiten especificar una lista de opciones disponibles. Son muy útiles cuando se tiene un conjunto de valores que puede tomar un tipo de variable determinado. Supongamos que tenemos un campo en una base de datos externa denominado **EstadoPublicacion**, que contiene los números 1, 2 o 3, que representan los estados en los que se puede encontrar una publicación de contenido de nuestra app web: **Borrador**, **EnRevision** y **Publicada**.

```
enum EstadoPublicacion {  
    Borrador,  
    EnRevision,  
    Publicada  
}
```

- Ahora, declaramos una variable para una nueva publicación, denominada “post” del tipo **EstadoPublicacion** y le asignamos el valor “EnRevision”:

```
let post: EstadoPublicacion = EstadoPublicacion.EnRevision;  
console.log(post);
```

Módulo 1. Intro a TypeScript

Enumeraciones (enum)

¿Qué valor se devuelve?

De forma predeterminada, los valores enum comienzan con un valor de 0

Para mostrar el nombre asociado a la enumeración, podemos usar el indexador proporcionado:

```
console.log(EstadoPublicacion[post]);
```

Módulo 1. Intro a TypeScript

any

- any es un tipo que puede representar cualquier valor de JavaScript sin restricciones. Esto puede ser útil si se espera un valor de una biblioteca de terceros o entradas de usuario en las que el valor es dinámico, ya que el tipo any permitirá volver a asignar distintos tipos de valores. Y, tal como se ha mencionado anteriormente, el uso del tipo any permite migrar gradualmente el código de JavaScript para usar tipos estáticos en TypeScript.
- Ejemplo:

```
let randomValue: any = 10;
randomValue = true;      // OK
randomValue = 'Mateo';   // OK
```

Módulo 1. Intro a TypeScript

① Importante

Recuerde que toda la comodidad de `any` se produce a costa de perder seguridad de tipos. La seguridad de tipos es uno de los principales motivos para usar TypeScript. Debe evitar el uso de `any` cuando no sea necesario.



Módulo 1. Intro a TypeScript

Tipos Unión

- Un tipo de **unión** describe un valor que puede ser uno de entre varios tipos. Esto puede ser útil cuando no tenga controlado un valor (por ejemplo, los valores de una biblioteca, una API o una entrada de usuario).
- Ejemplo:

```
let multiType: number | boolean;  
multiType = 20;          /* Valid  
multiType = true;       /* Valid  
multiType = "twenty";   /* Invalid
```

Módulo 1. Intro a TypeScript

Array

- Opción 1

```
let list: number[] = [1, 2, 3];
```

- Opción 2

```
let list: Array<number> = [1, 2, 3];
```

- Añadir un nuevo elemento

```
list.push(6);
```

Módulo 1. Intro a TypeScript

Tuple

- Error si intentamos añadir más de 2 elementos

```
var arr: [number, number] = [1,2,3];
```

```
app.ts:1:5 - error TS2322: Type '[number, number, number]' is not assignable to type '[number, number]'.  
Source has 3 element(s) but target allows only 2.
```

```
1 var arr: [number, number] = [1,2,3];
```

Aserción de Tipos

- Si necesita tratar una variable como un tipo de datos diferente, puede usar una **aserción de tipos**. Una aserción de tipos indica a **TypeScript** que ha realizado cualquier comprobación especial que necesite antes de llamar a la instrucción. Indica al compilador "**confía en mí, sé lo que estoy haciendo**".
- Las aserciones de tipos tienen dos formatos:

```
(randomValue as string).toUpperCase();
```

- La otra versión es la sintaxis de "corchetes angulares":

```
(<string>randomValue).toUpperCase();
```



Nota

`as` es la sintaxis preferida. Algunas aplicaciones de TypeScript, como JSX, pueden confundirse al usar `<>` para las conversiones de tipos.

Aserción de Tipos

- Ejemplo:

```
let randomValue: unknown = 10;

randomValue = true;
randomValue = 'Miguel';

if (typeof randomValue === "string") {
    console.log((randomValue as string).toUpperCase());      /* Returns MIGUEL en la consola.
} else {
    console.log("Error – A string was expected here.");      /* Returns an error message.
}
```

Restricciones de Tipos

- En el ejemplo anterior se muestra el uso de `typeof` en el bloque `if` para examinar el tipo de una expresión en tiempo de ejecución. A esto se le llama **restricción de tipos**.
- Puede usar las condiciones siguientes para descubrir el tipo de una variable:

Tipo	Predicate
<code>string</code>	<code>typeof s === "string"</code>
<code>number</code>	<code>typeof n === "number"</code>
<code>boolean</code>	<code>typeof b === "boolean"</code>
<code>undefined</code>	<code>typeof undefined === "undefined"</code>
<code>function</code>	<code>typeof f === "function"</code>
<code>array</code>	<code>Array.isArray(a)</code>

Módulo 1. Intro a TypeScript

LABORATORIO: **4 ejercicios sobre tipos en TypeScript**



01.TypeScript-Tipos

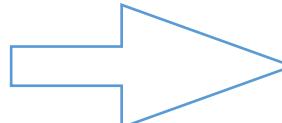
TIPOS AVANZADOS

Módulo 1. Intro a TypeScript

Custom Types

- Ejemplo 1

```
1 type Person = {  
2   firstName: string;  
3   lastName: string;  
4   age?: number;  
5 }  
6  
7 const person: Person = {  
8   firstName: 'Miguel'  
9 }
```



```
const person: Person = {  
  firstName: 'Miguel',  
  lastName: 'Campos'  
}
```

app.ts:7:7 - error TS2741: Property 'lastName' is missing in type '{ firstName: string; }' but required in type 'Person'.

Custom Types

- Ejemplo 2: tipo incorrecto

```
const person: Person = {  
  firstName: 'Miguel',  
  lastName: 3  
}
```

```
app.ts:3:5  
3   lastName: string;  
~~~~~  
'lastName' is declared here.
```

Módulo 1. Intro a TypeScript

interface

- Similar a type
- Ejemplo:

```
interface Person {  
    firstName: string;  
    lastName: string;  
    age?: number;  
}
```

```
const person: Person = {  
    firstName: 'Miguel',  
    lastName: 'Campos'  
}
```

Módulo 1. Intro a TypeScript

Diferencias entre type VS interface

Type	Interface
Es una colección de tipos de datos.	Es una forma de sintaxis.
Admite la creación de un nuevo nombre para un tipo.	Proporciona una forma de definir las entidades.
Tiene comparativamente menos capacidades.	Tiene comparativamente más capacidades.
No admite el uso de un objeto.	Es compatible con el uso de un objeto.
No se pueden usar varias declaraciones fusionadas.	Se pueden usar múltiples declaraciones fusionadas.
Dos tipos que tienen el mismo nombre provocan una excepción.	Dos interfaces que tienen el mismo nombre se fusionan.
No tiene propósitos de implementación.	Tiene un propósito de implementación.

Declaration Merging: merging interface

```
interface Box {  
    height: number;  
    width: number;  
}  
  
interface Box {  
    scale: number;  
}  
  
let box: Box = { height: 5, width: 6, scale: 10 };
```

Módulo 1. Intro a TypeScript

Clases

- Ejemplo:

```
class Persona {  
    nombre: string;  
  
    constructor(nuevoNombre: string) {  
        this.nombre = nuevoNombre;  
    }  
  
    decirMiNombre() {  
        console.log(this.nombre);  
    }  
  
}  
  
let p = new Persona('Miguel');
```

- Ya veremos en Angular la ventaja/desventaja de utilizar clases o interfaces

Herencia

- La herencia en Typescript se refiere a permitir crear nuevas interfaces que heredan o extienden el comportamiento de una anterior.

```
interface User {  
    firstName: string;  
    lastName: string;  
    email: string;  
}  
  
interface StaffUser extends User {  
    roles: Array<Role>  
}
```

LABORATORIO: Ejercicio herencia

Sobre la clase “Person” que hemos visto, implementar una clase “Empleado” que herede de esta, Aportando un nuevo atributo “sueldo”. Implementar en esta nueva clase un nuevo método “declarationRenta()” que indique por consola si el sueldo es superior a 10.000€ que debe presentar la declaración de IRPF, en otro caso no.

Módulo 1. Intro a TypeScript

Generics

- Podemos entender los genéricos como una especie de "plantilla" de código, mediante la cual podemos aplicar un tipo de datos determinado a varios puntos de nuestro código. Sirven para aprovechar código, sin tener que duplicarlo por causa de cambios de tipo y evitando la necesidad de usar el tipo "any".

```
function displayLog(valor: number) {  
    console.log(valor);  
}
```

```
function displayLogString(valor: string) {  
    console.log(valor);  
}
```

Módulo 1. Intro a TypeScript

Generics

- Solución 1: uso de tipo “any”

```
function display(valor: any): any {  
    console.log(valor);  
    return valor;  
}
```

Perdemos la ayuda del *intellisense*

Módulo 1. Intro a TypeScript

Generics

- Solución 2: uso de Generics

```
function display<T>(valor: T): T {  
    console.log(valor);  
    return valor;  
}  
  
display(1);  
display('Hola');
```

FUNCIONES

Funciones

- Las funciones son el bloque de construcción básico de cualquier aplicación, ya sean funciones locales, importadas de otro módulo o métodos en una clase. También son valores y, al igual que otros valores, TypeScript tiene muchas formas de describir cómo se pueden llamar a las funciones. Aprendamos a escribir tipos que describen funciones.
- Para empezar, al igual que en JavaScript, las funciones de TypeScript se pueden crear como una función con nombre o como una función anónima.

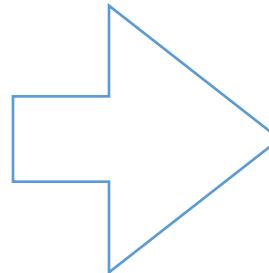
```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function (x, y) {
    return x + y;
};
```

Funciones flecha (arrow functions)

- Nos permite una sintaxis más reducida y clara del código:

```
var sum = function (x, y) {  
    return x + y;  
}
```



```
let sum = (x: number, y: number) => x + y;
```

Function Type Expressions

- Este tipo de funciones son semánticamente similares a las arrow functions:

```
function greeter(fn: (a: string) => void) {  
  fn("Hello, World");  
}  
  
function printToConsole(s: string) {  
  console.log(s);  
}  
  
greeter(printToConsole);
```

Parámetros opcionales

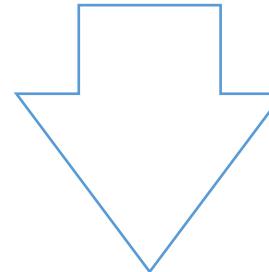
- Podemos marcar en las funciones parámetrosopcionales, haciendo uso del operador ?:

```
function f(x?: number) {  
    // ...  
}  
f(); // OK  
f(10); // OK
```

Parameter Destructuring

- Podemos marcar en las funciones parámetros opcionales, haciendo uso del operador ?:

```
function sum({ a, b, c }: { a: number; b: number; c: number }) {  
  console.log(a + b + c);  
}
```



```
type ABC = { a: number; b: number; c: number };  
function sum({ a, b, c }: ABC) {  
  console.log(a + b + c);  
}
```

Functions: reduce, filter, find

- **reduce()**: este método aplica una función simultáneamente a dos valores de un array (de izquierda a derecha) para reducirlo a un solo valor.
- Sintaxis:

```
array.reduce(callback[, initialValue]);
```

- Parámetros:
 - **callback** – función a ejecutar por cada valor del array.
 - **initialValue** – valor inicial que se aplica a la primera invocación del callback.
- Ejemplo:

```
var total = [0, 1, 2, 3].reduce(function(a, b){ return a + b; });
console.log("total is : " + total );
```

- Resultado:
 - El resultado es un valor único, en este ejemplo sería **6**.
 - Si hubiéramos aplicado un **initialValue** de **5**, el resultado hubiera sido **11**.

Functions: reduce, filter, find

- **filter()**: este método crea un nuevo array con todos los elementos que cumplan la condición implementada por la función dada.
- Sintaxis:

```
array.filter(callback);
```

- Ejemplo:

```
const technologies = [  
  {name: "Python", type: "IA"},  
  {name: "Angular", type: "Frontend"},  
  {name: "Vue", type: "Frontend"},  
  {name: "React", type: "Frontend"},  
  {name: "Nodejs", type: "Backend"},  
  {name: "Nestjs", type: "Backend"},  
  {name: "Docker", type: "Devops"},  
  {name: "Kubernetes", type: "Devops"}  
]  
  
technologies.filter(x => x.type == "Frontend");
```

- Resultado:

```
▼ (3) [ {...}, {...}, {...} ] ⓘ  
▶ 0: {name: "Angular", type: "Frontend"}  
▶ 1: {name: "Vue", type: "Frontend"}  
▶ 2: {name: "React", type: "Frontend"}  
  length: 3  
▶ __proto__: Array(0)
```

Módulo 1. Intro a TypeScript

Functions: reduce, filter, find

- `find()`: este método devuelve el valor del primer elemento del array que cumple la función de prueba proporcionada.
- Sintaxis:

```
array.find(callback);
```

- Ejemplo:

```
console.log(technologies.find(x => x.type == "Frontend"));
```

- Resultado:

```
app.ts:49:26 - error TS2550: Property 'find' does not exist on type '{ name: string; type: string; }[]'. Do you need to change your target library? Try changing the 'lib' compiler option to 'es2015' or later.
```

- Solución: indicar que queremos compilar con el estándar es6

```
tsc app.ts --watch --target es6
```

- "ES3" (default)
- "ES5"
- "ES6"/"ES2015"
- "ES2016"
- "ES2017"
- "ESNext"