

# TypeScript

---

# Introducción

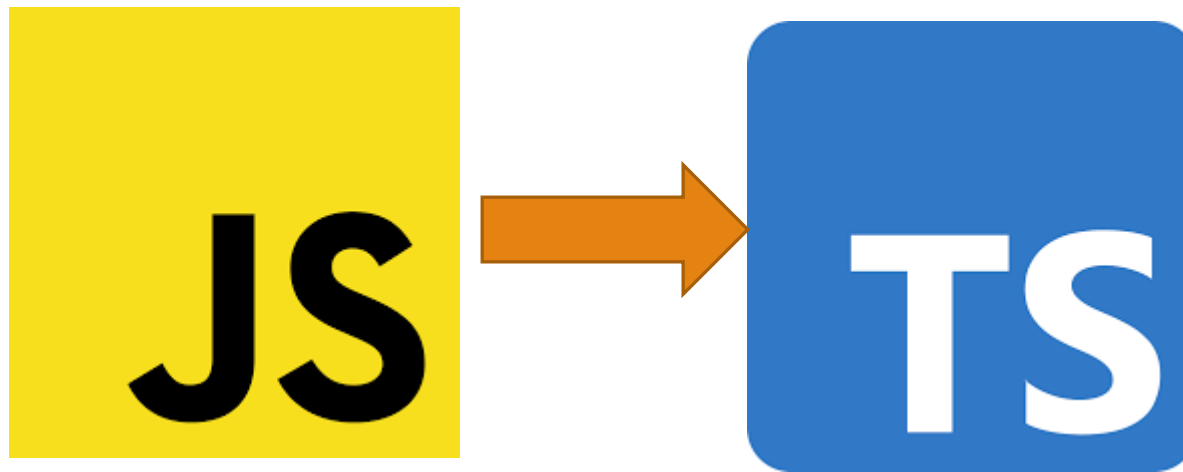
---

# TypeScript

---

## ❑ ¿Qué es Typescript?

- ❑ Superset tipado de Javascript.
- ❑ Necesita ser transformado (transpilado) a JS para su ejecución.
  - ❑ JS === TS.
  - ❑ TS !== JS.



# TypeScript

---

## ❑ Ventajas - Seguridad

❑ Gracias al análisis estático de código y al uso de datos tipados, minimizamos bugs en ejecución.

```
function sum(a:number, b:number): number {  
    return a+b;  
}  
  
console.log(sum('10' + 30));
```

❑ El compilador devuelve:

```
$ tsc sample.ts sample.ts(5,13): error TS2346: Supplied parameters  
do not match any signature of call target.
```

# TypeScript

---

## ❑ Ventajas - Legibilidad

- ❑ Conocer los tipos de datos esperados, ayudan a hacer el código más legible
- ❑ Incluso cuando se usan nombre de variables poco claros.

```
function initialize(s: Vehicule, d: Date) {  
    s.setStartDate(d);  
    s.checkEngine();  
    return s;  
}
```

# TypeScript

---

## ❑ Ventajas - Tooling

- ❑ Existen editores con una gran integración con Typescript.
- ❑ Facilitan las ayudas inline y el autocompletado.

```
const a = [0, 1, 2, 3, 4];  
a.splice(1, )
```

```
splice(start: number, deleteCount?: number):  
number[]
```

The number of elements to remove.

Removes elements from an array and, if necessary, inserts new elements in their place, returning the deleted elements.

# TypeScript

---

## ❑ Ventajas - Tooling



❑ Detección de errores mientras escribimos código.

```
function sum(a:number, b:number): number {  
    return a+b;  
}  
console.log(sum('10' + 30));
```

# TypeScript

---

## ❑ Historia

- ❑ Es un lenguaje de programación de código abierto, desarrollado y mantenido por Microsoft.
- ❑ Se publicó en Octubre de 2012 (versión 0.8), después de 2 años de desarrollo interno.
- ❑ Actualmente (Julio 2021) Versión 4.3
- ❑  El compilador de TS está escrito en TS 



# TypeScript

---

☐ ¿Qué hace falta?

☐ Node.js (es recomendable con nvm o similar)

☐ Instalar typescript:

```
npm install -g typescript
```

☐ Ya disponemos de tsc en nuestra terminal:

```
$ tsc sample.ts
```

# TypeScript

---

## □ Ejemplo

```
# sample.ts
function sum(a:number, b:number): number {
  return a+b;
}
console.log(10 + 30);
```



```
# sample.js
function sum(a, b) {
  return a+b;
}
console.log(10 + 30);
```

# TypeScript

---

## ❑ TS en Node.js

- ❑ El paquete `ts-node` permite ejecutar typescript directamente sin necesidad de transpilar previamente:

```
npm install -g ts-node
```

- ❑ Una vez instalado, podremos ejecutar directamente:

```
$ ts-node sample.ts  
40
```

# TypeScript

---

## ☐ Integración en Herramientas

- ☐ Es poco habitual utilizar directamente tsc para transpilar nuestros código TS.

## ☐ Integraciones en las principales herramientas de build:

- ☐ Webpack (ts-loader)

- ☐ Browserify (tsify)

- ☐ Babel (babel-preset-typescript)

- ☐ Gulp (gulp-typescript)

- ☐ ...

# TypeScript

---

## ☐ IDEs

### ☐ Visual Studio Code

- ☐ Soporte nativo. Plugins

- ☐ Opensource (github)

### ☐ Sublime Text (+ plugin)

### ☐ Atom (+ plugin)

### ☐ WebStorm (not free / not open)

# Tipos de Datos

---

# TypeScript

---

## ❑ Core Types

## ❑ TS soporta los tipos de datos definidos en JS:

❑ boolean · number · string · array · object

## ❑ Añadiendo además:

❑ tuple · enum · void · null · undefined · never · any

# TypeScript

---

## □ Boolean

```
let state: boolean = true;  
state = null; // nullable!
```

## □ Number

```
// Como Javascript, podemos tener enteros o flotantes.  
let pi: number = 3.14;  
const uno: number = 1;  
// también en hexadecimal  
let hexadecimal: number = 0xFF; // 255
```

## □ Se incluye además notación binaria y octal (ES2015)

```
let binary: number = 0b10; // 2  
let octal: number = 0o1232; // 666
```



# TypeScript

---

## □ String

```
let name: string = 'Jon';  
let name2: string = "Maite";
```

## □ Se incluye soporte para *template strings* (ES2015):

```
console.log('Hola Mundo!!!!!!!!!!');  
var n1="pepe"  
var n2='juan'  
let gretting: string = `Hola ${n1}!  
Hola ${n2}!  
`;  
console.log(gretting)
```

```
Hola Mundo!!!!!!!!!!  
Hola pepe!  
Hola juan!  
[WDS] Live Reloading enabled.  
|
```

# TypeScript

---

## ❑ array

- ❑ Un array se puede declarar usando el tipo de dato que contendrá el array:

```
let names: string[] = [];  
names.push('Jon');  
names.push('Ane');
```

- ❑ O bien usando el tipo genérico **Array**, especificando el tipo de sus elementos:

```
let names: Array<string> = [];  
names.push('Jon');  
names.push('Ane');
```

# TypeScript

---

## □ tuple

- Este tipo de dato es un subtipo de un array; concretando número y tipo de sus elementos:

```
let point: [number, number, string];  
  
// Asignando valores:  
point = [43.2603479, -2.9334110, 'Bilbao']; // OK  
  
// Asignando valores:  
point = ['Bilbao', 43.2603479, -2.9334110]; // Error
```

# TypeScript

---

## □ Enum (I)

- Un tipo de dato enum es la manera de usar alias amigables para un set de valores numéricos.

```
enum Color { Rojo, Verde, Azul};  
let c: Color = Color.Rojo;  
console.log('>> ' + c) // >> 0
```

- Está permitido usar índices arbitrarios:

```
enum Color { Rojo = 43, Verde, Azul = 50};  
console.log('>> ' + Color.Rojo) // >> 43  
console.log('>> ' + Color.Verde) // >> 44  
console.log('>> ' + Color.Azul) // >> 50  
}
```

# TypeScript

---

## ❑ Enum (II)

- ❑ Es posible recuperar el identificador a partir del índice:

```
enum Color { Rojo = 43, Verde, Azul = 50};  
console.log('>> ' + Color[43]) // >> Rojo
```

- ❑ También es posible usar enums de tipo string (desde la versión 2.4):

```
enum Color { Rojo = 'RED', Verde = 'GREEN', Azul = 50};  
console.log('>> ' + Color.Rojo) // >> RED
```

# TypeScript

---

## ❑ void

- ❑ El tipo void es usado para especificar la ausencia de tipo; muy útil para funciones que no devuelven valores.

```
function log(msg:string): void {  
    console.log(msg);  
}
```

- ❑ Se pueden declarar variables de tipo void pero solo podrán tener asignado null o undefined:

```
let x:void;  
x = null;  
x = undefined;  
x = false; // COMPILER ERROR!
```

# TypeScript

---

## ❑ Object

- ❑ Un tipo object representa cualquier tipo no primitivo: number, string, boolean, null, or undefined.

```
let a:object = {};  
a.toString(); // OK  
a.toMagicalString(); // COMPILE ERROR
```

- ❑ Su diferencia con any es que object SI comprueba tipado:

```
let a:any = {};  
a.toString(); // OK  
a.toMagicalString(); // OK >> RUNTIME ERROR
```

# Interfaces

---



# TypeScript

---

## ❑ Interfaces

- ❑ Una interfaz especifica una lista de campos y funciones con tipos concreto (o any), con el propósito de describir un tipo de dato de un objeto.

```
interface Point {  
    coordX: number;  
    coordY: number;  
    label: string;  
};
```

# TypeScript

---

## ❑ Interfaz de clase

❑ Una interfaz podrá ser implementada desde una clase:

```
class Coordinada implements Point {  
    public coordX: number;  
    public coordY: number;  
    public label: string;  
  
    constructor(x: number, y: number, label: string) {  
        this.coordX = x;  
        this.coordY = y;  
        this.label = label;  
    }  
}  
  
const bio = new Coordinada(43.2603479, -2.9334110, 'Bilbao');
```

# TypeScript

---

## ❑ Interfaz de tipo

❑ O también, una interfaz podrá especificar un tipo de dato:

```
const Bilbao: Point = {  
  coordX: 43.2603479,  
  coordY: -2.9334110,  
  label: 'Bilbao'  
};
```

# TypeScript

---

## ❑ Propiedades opcionales

### ❑ Es posible definir propiedades opcionales:

```
interface Point {  
  coordX: number;  
  coordY: number;  
  label: string;  
  description?: string;  
};
```

### ❑ Ésta tendrá el valor undefined en ejecución:

```
const c:Point = {  
  coordX: 43.2603479,  
  coordY: -2.9334110,  
  label: 'Bilbao'  
};  
console.log(c.description === undefined); //true
```

# TypeScript

---

## ❑ Propiedades readonly

❑ Es posible definir propiedades como solo lectura; es decir inmutables.

```
interface Point {  
    readonly coordX: number;  
    readonly coordY: number;  
    label: string;  
};  
  
const c:Point = {coordX: 43.2603479, coordY: -2.9334110, label:  
    'Bilbao'};  
c.coordX = 44.2603479; // COMPILATION ERROR!!
```

# TypeScript

---

## ❑ Interfaces anidadas

- ❑ Es posible anidar distintos tipos de interfaces siempre que estén previamente definidos
- ❑ Una vez definida una interfaz, ésta pasa a ser un tipo de datos (re)usable y exportable en nuestra aplicación

```
interface Geo {  
  x: number;  
  y: number;  
}  
  
interface City {  
  coords: Geo;  
  label: string;  
}
```

# TypeScript

---

## ❑ Function Type

- ❑ Es posible crear interfaces que describen funciones: Se describe la lista de parámetros, y su valor de retorno.

```
interface GreetFunc {  
  (name: string): string;  
}
```

```
interface Greet {  
  lang: string;  
  greet: GreetFunc;  
}
```



```
const p:Greet = {  
  lang: 'es',  
  greet: (nombre) => `Hola ${nombre}`  
}
```

# TypeScript

---

## ❑ Extendiendo interfaces

- ❑ Una interfaz puede extender otra interfaz, heredando directamente sus definiciones:

```
interface Person {  
  name: string;  
  age: number;  
}  
  
interface Cyborg extends Person {  
  serialNumber: number;  
}  
  
const p: Cyborg = {  
  name: 'Robocop',  
  age: 35,  
  serialNumber: 0x7F5B63  
}
```







# TypeScript

---

## ❑ Integración en IDE

### ❑ Ventajas de usar interfaces:

- ❑ Mejora la legibilidad
- ❑ Ayuda en la codificación
- ❑ Evita typos

```
const p: Cyborg = {  
  name: 'Robocop',  
  age: 35,  
  ...  
}  
   serialNumber (property) Cyborg.serialNumber: number  
   #endregion  
   #region  
   class
```

# Classes

---

# TypeScript

---

## ❑ Classes

- ❑ TS evoluciona el sistema de clases desarrollado con ES2015 (OO).
- ❑ Nos olvidamos del sistema basado JS de prototype y funciones instanciables.

```
class Vehiculo {  
  marca: string;  
  constructor(marca: string) {  
    this.marca = message;  
  }  
  arrancar() {  
    return "flecha";  
  }  
}
```

# TypeScript

---

## □ Herencia

- Se implementa un sistema clásico de herencia, de manera que un clase puede extender otra.

```
class Coche extends Vehiculo {  
  
    constructor(marca: string) {  
        super(marca); // invocamos método del padre  
    }  
    arrancar() {  
        return "X";  
    }  
    radio() {  
        return "Z";  
    }  
}
```

# TypeScript

---

## ❑ Modificadores de ámbito (public)

❑ Por defecto todos los método y propiedades serán de acceso público:

```
class Coche extends Vehiculo {  
  constructor(marca: string) {  
    super(marca); // invocamos método del padre  
  }  
  
  arrancar() {  
    return "brummm";  
  }  
}  
  
const coche1 = new Coche("kia");  
  
console.log(coche1.marca); // kia  
  
console.log(coche1.radio()); // radio
```

# TypeScript

---

## ❑ Modificadores de ámbito (private)

- ❑ Si se especifica, una propiedad o método privado solo podrá invocarse desde su clase contenedora.

```
class Coche extends Vehiculo {  
    private velocidad: number;  
  
    constructor(marca: string) {  
        super(marca); // invocamos método del padre  
    }  
    arrancar() {  
        this.velocidad = 10;  
        return "Brummm";  
    }  
    private frenar() {  
        return "Frenando";  
    }  
}
```

# TypeScript

---

## ❑ Modificadores de ámbito (private)

❑ La principal ventaja es que el error ocurrirá en tiempo de compilación:

```
const coche1 = new Coche("kia");  
coche1.arrancar(); // OK  
  
console.log(coche1.velocidad); // ERROR en compilación  
  
coche1.frenar(); // ERROR en compilación
```

Property 'velocidad' is private and only accessible within class 'Coche'.  
Property 'frenar' is private and only accessible within class 'Coche'.

# TypeScript

---

## ❑ Modificadores de ámbito (protected)

- ❑ Una propiedad o método `protected` es similar a los declarados como `private`, pero podrá ser accedido desde clases derivadas

```
class Vehiculo {  
    private bastidor;  
    protected serialNumber;  
    /* .... */  
}  
class Coche extends Vehiculo{  
    /* .... */  
    public getInfo() {  
        return {  
            sn: this.serialNumber,  
            bastidor: this.bastidor  
        }  
    }  
}
```



# TypeScript

---

- ❑ **Modificadores de ámbito (protected)**
- ❑ **El error será también lanzado desde el transpilador, al intentar acceder a la variable bastidor.**

```
const coche1 = new Coche("kia");  
  
coche1.getInfo(); // ERROR en compilación
```

Property 'bastidor' is private and only accessible within class 'Vehiculo'.

# TypeScript

---

## ❑ Modificador de acceso (readonly)

- ❑ Una propiedad especificada como readonly deberá ser inicializada en declaración o en el constructor.

```
class Entrada {  
    readonly precio:number = 10;  
    readonly pelicula: string;  
  
    constructor(pelicula) {  
        this.pelicula = pelicula;  
    }  
}  
  
const e1 = new Entrada('Deadpool 2');  
e1.pelicula = 'Han Solo'; // ERROR en compilación  
e1.precio = 5; // ERROR en compilación
```

# TypeScript

---

## ☐ Parámetros del constructor

☐ Si utilizamos uno de los modificadores de accesibilidad en el constructor, estos parámetros pasan a ser propiedades de la clase:

☐ public

☐ private

☐ protected

☐ readonly

# TypeScript

---

## ❑ Parámetros del constructor

❑ Si utilizamos uno de los modificadores de accesibilidad en el constructor, estos parámetros pasan a ser propiedades de la clase:

❑ **public**

❑ **private**

❑ **protected**

❑ **readonly**

```
class Persona {  
  constructor(  
    public nombre:string,  
    private peso: number,  
    readonly ojos: string,  
    edad) {}  
}  
  
const p = new Persona("han solo", 80, "marrones", 20);  
console.log(p.nombre);  
console.log(p.ojos);  
console.log(p.peso); // Property 'peso' is private and only accessible  
within class 'Persona'.  
console.log(p.edad); // Property 'edad' does not exist on type 'Persona'.
```

# TypeScript

---

## □ getter / setter

□ Es posible definir métodos setter y/o getters:

□ El uso de getters/setter obliga a compilar mínimo para ES5 (tsc --target ES5).

```
class Persona {  
  constructor(private nombre, private apellido) {}  
  get identificacion() {  
    return `${this.nombre} ${this.apellido}`;  
  }  
  set identificacion(identificacion) {  
    this.nombre = identificacion.split(' ')[0];  
    this.apellido = identificacion.split(' ')[1];  
  }  
}  
  
const p = new Persona('Han', 'Solo');  
console.log(p.identificacion); // Han Solo  
p.identificacion = "Luke Skywalker";  
console.log(p.identificacion); // Luke Skywalker
```

# TypeScript

---

## ❑ Propiedades estáticas

❑ TS nos permite declarar propiedades (o métodos) estáticas en nuestras clases (públicas, protected o privadas):

```
class Persona {  
  static tipo = 'humano';  
  constructor(public nombre: string) {}  
  
  getInfo() {  
    return `Soy ${this.nombre}, ${Persona.tipo}.`;  
  }  
}  
  
const p = new Persona('Jabba')  
console.log(p.getInfo()); // Soy Jabba, humano  
console.log(Persona.tipo); // humano
```

# TypeScript

---

## ❑ Clases abstractas

❑ TS permite definir una clase como abstract:

❑ Permite definir propiedades/métodos como abstractos, e implementar otros.

```
export abstract class Base {  
  
    abstract greet():string;  
  
    adios():void {  
        console.log("agur!")  
    }  
}
```

# TypeScript

---

## ❑ Clases abstractas

### ❑ TS permite definir una clase como abstract:

```
class Saludo extends Base {  
  
    greet():string {  
        return "hola";  
    }  
}  
  
const s = new Saludo();  
s.greet(); // hola!  
s.adios(); // agur!
```



# Funciones

---

# TypeScript

---

## ❑ Functions

- ❑ Las funciones en TS funcionan de manera idéntica a JS:
  - ❑ tienen un contexto de ejecución
  - ❑ tienen acceso a su contexto padre
  - ❑ Y éstas, pueden ser definidas con nombre o anónimas:

```
function f1() {  
  console.log('f1!');  
}  
const f1 = function() { console.log('f1!');}
```

# TypeScript

---

## □ Functions

### □ Tipando funciones

- Se pueden tipar tanto los parámetros, como lo que devuelve nuestra función:

```
function suma(x:number, y: number): number {  
    return x+y;  
}
```

### □ Y también se puede crear un tipo función:

```
const x: (sumando1: number, sumando2: number) => number =  
    function(x:number, y:number) { return x+y; };
```

# TypeScript

---

## ❑ Parámetros opcionales

- ❑ Se pueden declarar los parámetros como opcionales:
- ❑ JS, soporta por defecto parámetros opcionales; lo bueno de TS es que quién lee el código, espera que puedan ser opcionales.

```
function saluda(aQuien?:string): void {  
  console.log(`Hola ${aQuien || 'mundo'}`);  
}  
  
saluda("Pepito"); // Hola pepito  
saluda(); // Hola mundo
```

# TypeScript

---

## ❑ Parámetros por defecto

### ❑ Se pueden declarar los parámetros por defecto:

```
function saluda(aQuien = 'mundo'): void {  
    console.log(`Hola ${aQuien}`);  
}
```

```
saluda("Pepito"); // Hola pepito  
saluda(); // Hola mundo  
saluda(undefined); // Hola mundo
```

### ❑ Es decir, que valor deben adquirir si se recibe un undefined:

# TypeScript

---

## □ Parámetros Rest

□ Se permiten recibir "n" parámetros en un array:

```
function sum(...nums: number[]): number {  
    return nums.reduce((acc, num) => acc+num, 0);  
}  
console.log(sum(2,2,2,2,2)); // 10
```

```
function operation(type: string, initial: number, ...nums: number[]): number {  
    if (type === 'sum') return nums.reduce((acc, num) => acc+num, initial);  
    else if (type === 'sub') return nums.reduce((acc, num) => acc-num, initial);  
}  
console.log(operation('sub',10,2,2,1)); // 5
```

# TypeScript

---

## ❑ Parámetros De-structuring

- ❑ Es posible des-estructurar parámetro a parámetro, directamente en el cuerpo de la función:

```
interface Person { name: string, age: number }

function saluda({ name, age}: Person) {
  console.log(`Hola ${name}. Tienes ${age} años`);
}

const p: Person = { name : 'Pepito', age: 20};
saluda(p); // Hola Pepito. Tienes 20 años.
```

# TypeScript

---

## ❑ Overload (I)

❑ Typescript permite de manera bastane "manual" hacer function overload.

```
let currentVolumen = 50;
const levels = { 'high': 100, 'medium': 50, 'low': 10};
function changeVolumen(x:any): any {

    if (typeof x === "string") {
        return currentVolumen-levels[x];
    }

    else if (typeof x == "number") {
        return currentVolumen-x;
    }
}
```



# TypeScript

---

## ❑ Overload (II)

❑ TS nos permite tipar todos los posibles casos:

```
let currentVolumen = 50;
const levels = { 'high': 100, 'medium': 50, 'low': 10};

function changeVolumen(x:number): number;
function changeVolumen(x:string): number;
function changeVolumen(x:any): any { // <- NOT CHECKED!

    if (typeof x === "string") {
        return currentVolumen-levels[x];
    }

    else if (typeof x == "number") {
        return currentVolumen-x;
    }
}
```

```
console.log(changeVolumen(25)); // 25

console.log(changeVolumen('high')); // 100

console.log(changeVolumen({x:99})); // COMPILATION
ERROR!!
```