Introducción a TypeScript

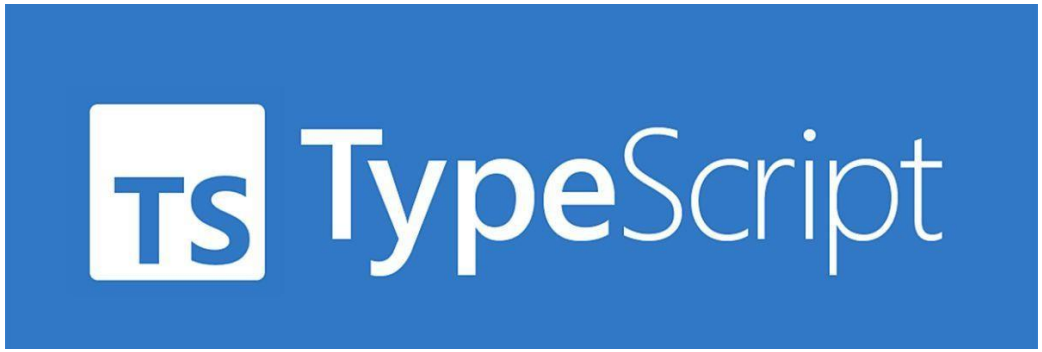
UD2: Desarrollo de aplicaciones basado en componentes con Angular

Objetivos de aprendizaje

- Comenzar a utilizar TypeScript y comprender las diferencias básicas con Javascript
- Aprender a utilizar y declarar tipos básicos, funciones y objetos

Indice

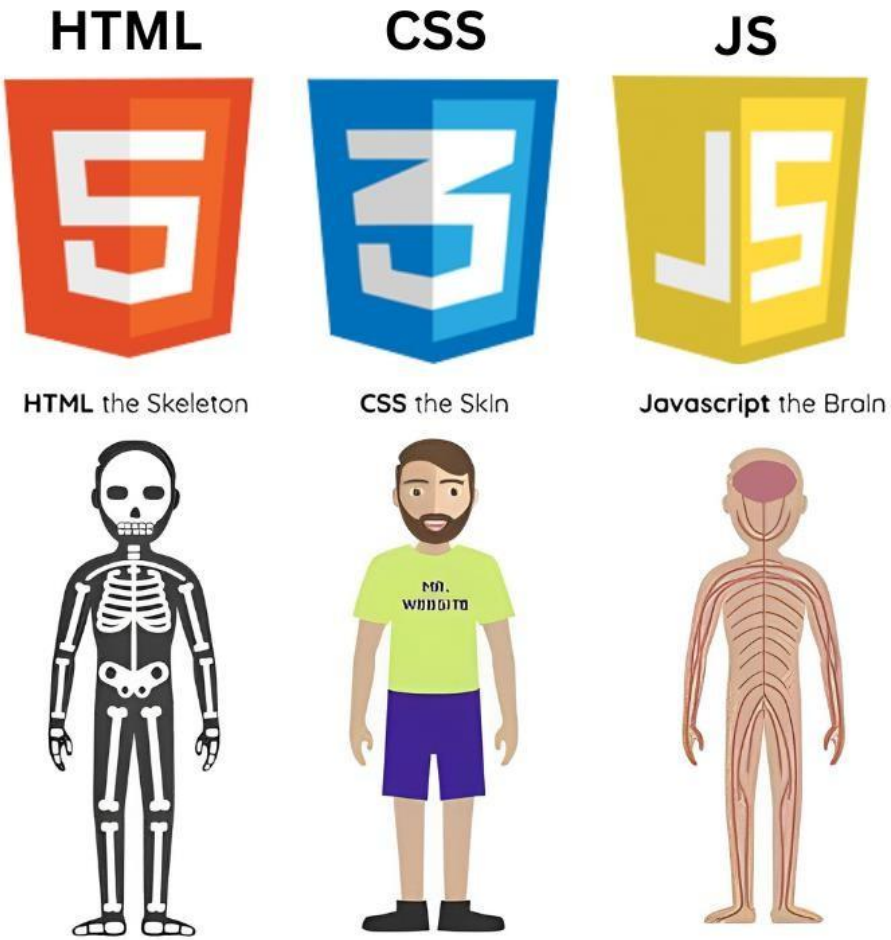
- Introducción a Typescript
- Tipado básico
- Funciones y objetos



Introducción a Typescript

Indice

Recordemos…


HTML
CSS
JS
HTML the Skeleton
CSS the Skin
Javascript the Brain

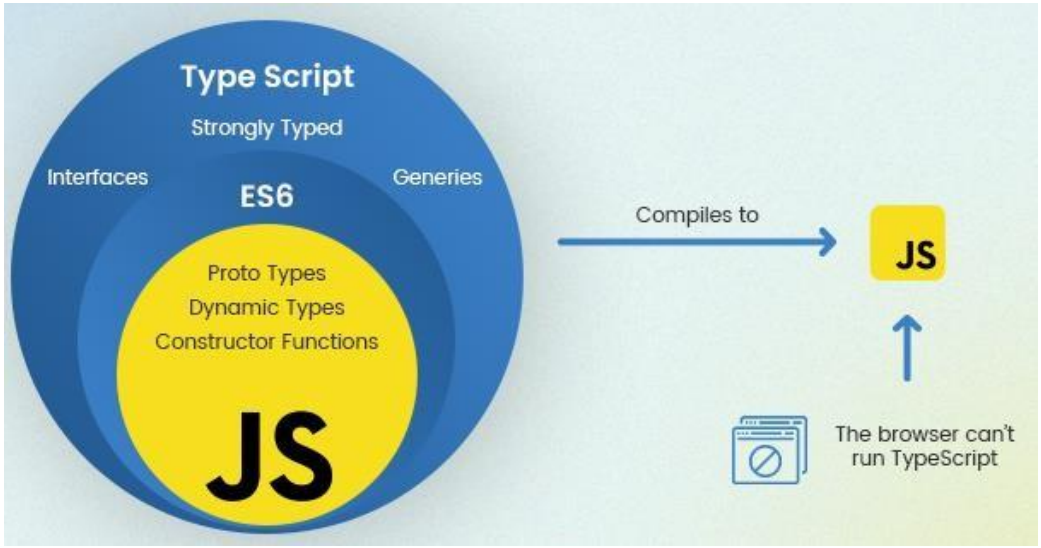
Javascript (JS) y TypeScript (TS)

- Es un lenguaje que dota a JS de características adicionales
- Hace que el código tenga menos errores, sea más sencillo y fácil de probar
- Introducido por MS en 2012 pero adoptado por Google como **lenguaje por defecto para Angular**

Todo el código escrito en JS es valido para TS pero no al revés

Javascript (JS) y TypeScript (TS) (II)

- Los navegadores no entienden Typescript, luego hay que hacer un paso a Javascript llamado **transpilación**



COMPILAR – Pasar de un lenguaje a otro de más bajo nivel de abstracción

TRANSPILAR – Pasar de un lenguaje a otro de nivel de abstracción similar

Valoración

Ventajas de Typescript	Desventajas de Typescript
Tipado estático	Sensación de escribir más código
Seguridad	Curva de aprendizaje mayor
Legilidad	No es 100% fiable (pues al final ejecutamos Javascript)

Requisitos

- Node 
- npm 

Instalando Node.js

- Descargamos el instalador desde la página de descargas de Node
  - Elegimos versiones LTS (más estables) frente a las actuales

- Comprobamos si ya tenemos Node instalado

```
node --version
```

```
node script_ejemplo.js
```

npm (Node Package Manager)

- Permite obtener librerías/paquetes de manera simple.
  - A las librerías que utilizo en mi aplicación las llamamos dependencias
- Por defecto, las dependencias se instalan en una carpeta local (`node_modules`) dentro de nuestro proyecto
  - También podemos indicar que queremos instalarlos de manera global para todos los proyectos
- Comprobamos si ya tenemos npm instalado

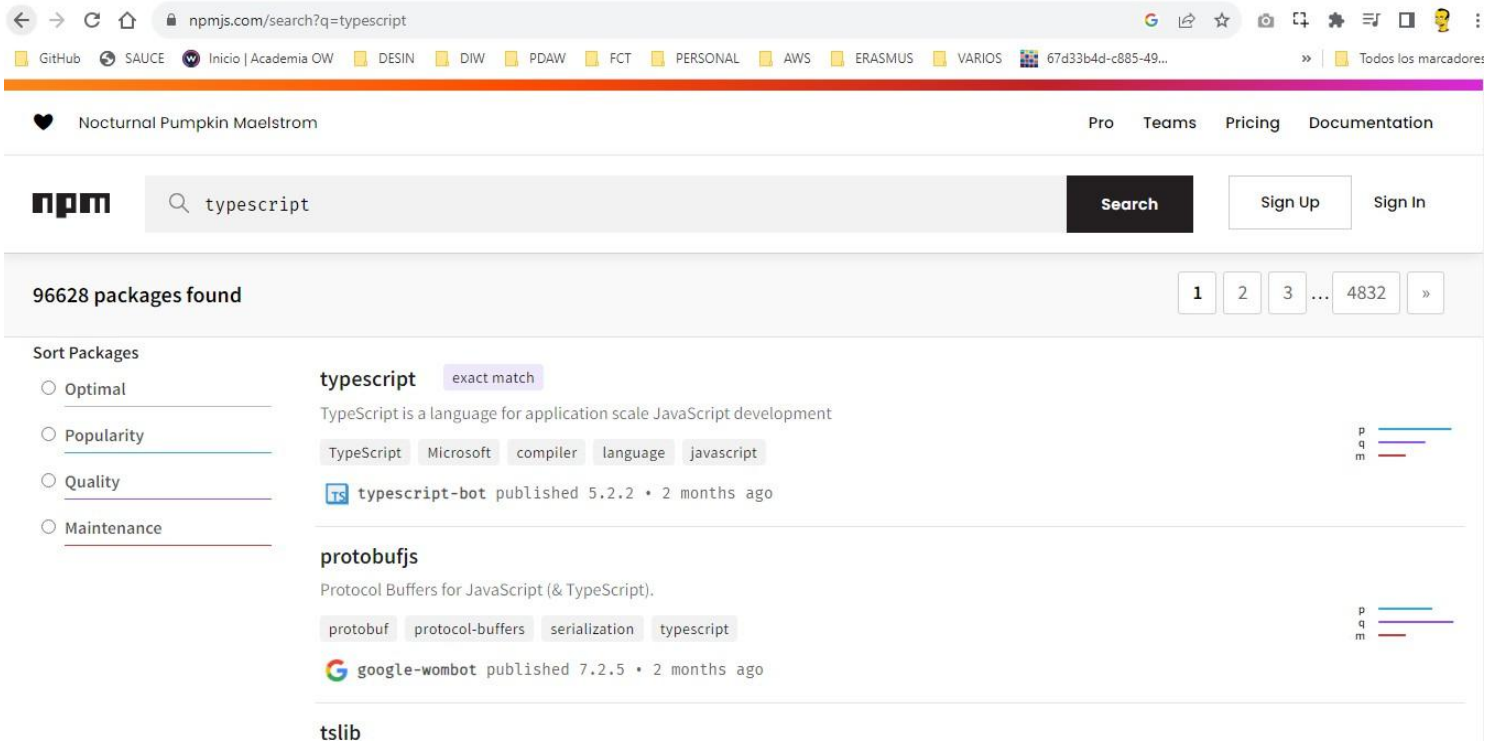
```
npm --version
```

- Actualización

```
npm install -g npm@latest
```

Buscador de paquetes npm

- https://www.npmjs.com/



Instalación de Typescript

- Instalación a nivel de mi proyecto

```
npm i typescript
```

- La dependencia se mete en la carpeta node_modules
  - Debemos introducir dicha carpeta en el .git_ignore
- Instalación a nivel global (para todo el equipo)

```
npm install -g typescript
```

Git y npm

- El fichero `.gitignore` nos permite determinar qué carpetas queremos excluir del versionado de código
  - Se coloca en la raíz del proyecto
- Es importante incluir el archivo `node_modules` para evitar que las dependencias se suban a GitHub

```
node_modules
```

`.gitignore`

Cuando descargamos el proyecto a otro equipo podemos instalar de nuevo las dependencias con `npm install`

Ejecución de Typescript

- Ejecutamos Typescript con el comando tsc
- Si hemos instalado a nivel de proyecto

npx es una herramienta de npm que permite ejecutar paquetes sin necesidad de instalarlos globalmente (o incluso sin instalarlos, los descarga automáticamente)

- Si hemos instalado a nivel global

Inicialización de Typescript

```
tsc --init
```

- Crea un fichero llamado `tsconfig.json` que contiene la configuración de la compilación
- Podemos configurar el directorio de salida, cómo de estrictos podemos ser, si queremos sacar los comentarios, etc..
- Ver opciones

Posibles problemas

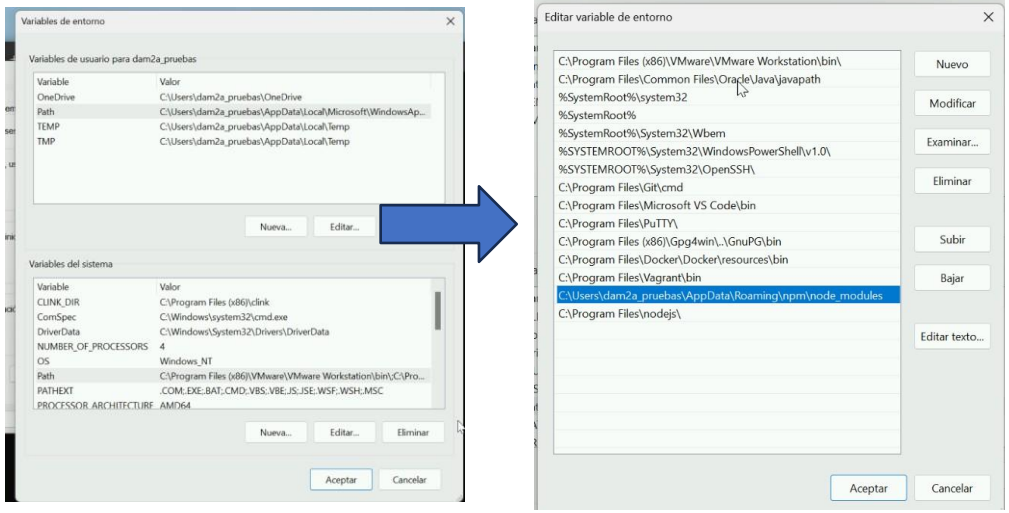
- Si el comando `tsc` no se detecta podríamos necesitar añadir la ruta de npm al PATH

```
npm root -g
```

Obtenemos la ruta donde está instalado npm

- Configuración avanzada del sistema > Variables de entorno > PATH
  - Añadimos la ruta anotada

El PATH es una variable de Windows que indica donde tenemos que buscar los comandos que ejecutamos



Transpilación

`hola_mundo.ts`

`tsc hola_mundo.ts`

`hola_mundo.js`

`tsc` `Transpila todos los ficheros ts`

Posibles problemas

- A veces Typescript almacena configuraciones en caché

```
tsc --project tsconfig.json
```

Fuerza a Typescript a usar el archivo especificado evitando cualquier valor en caché

Modo watch (vigilante)

- Busca cambios en el código y compila automáticamente

```
tsc hola_mundo.ts --watch
```

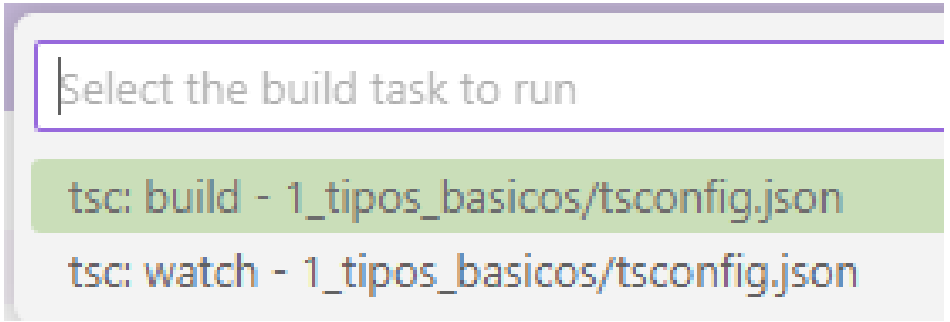
- Si queremos hacerlo para todo el código de mi proyecto

```
tsc --watch
```

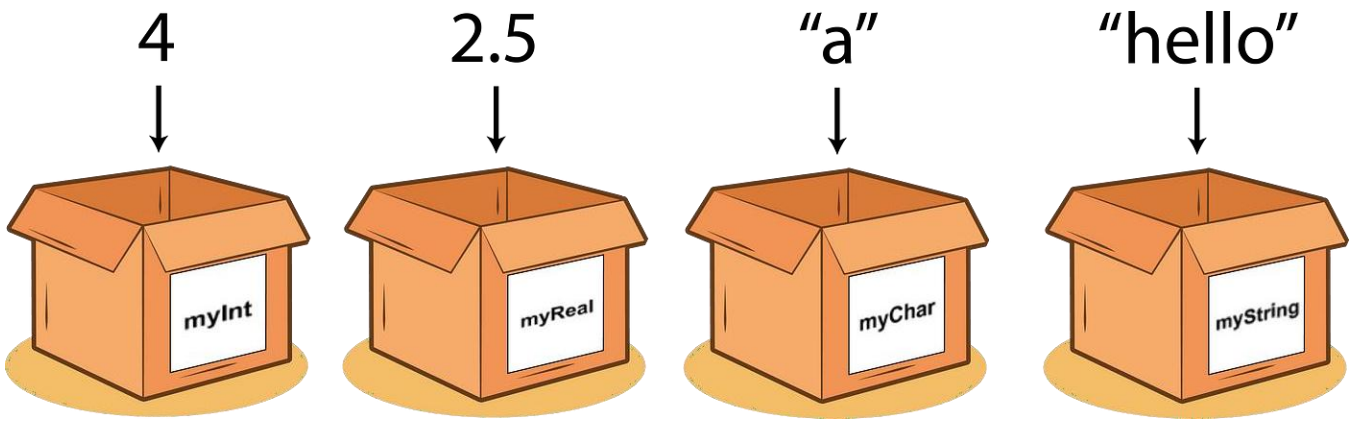
Typescript y VSCode

- VSCode integra tsc a través de un gestor de compilación (`Build Task`)

`Control+Shift+B`



`Necesitamos tener Typescript inicializado`



Tipado básico

Indice

Declaración de variables

```
var|let|const variable:tipo=valor
```

- Las variables <u>deben declararse antes de usarse</u>
  - Aunque podemos declarar una variable sin asignar tipo y este se detecta automáticamente al inicializar.
- Se detectan:
  - Errores de declaración (no podemos usar variables no declaradas)
  - Errores de cast (no podemos asignar valores de un tipo distinto

Tipos básicos

Tipo	Valores	Ejemplo
boolean	true | false	`let flag:boolean=false`
number	Número entero o real	`let edad:number=23;` `let dinero:number=3.42`
string	Cadena entre comillas simples, dobles o invertidas	`let nombre:string=`Manuel``
array	[variable|valor1, variable|varlor2]	`let nombres:string[];`
any	Cualquier valor	`let nombres:any[]`

Tipos básicos: Ejemplo

```ts
let color:string   = "rojo";
color="verde";

let nombre:string  = "Juan";
let edad:number = 25;

let texto:string = `Hola, ${nombre} (${edad})`;

let texto2:string = `${1+2}`;
```

```ts
let listado:number[] = [1,2,3,4,5];
let nuevoListadoo: [string,number]= ["Juan", 25];
```

Uso de any

- Se utiliza cuando no tenemos claro el tipo de dato que vamos a usar.

```
let noLoSe:any = "hola";
noLoSe = 10;
noLoSe = true;


let listaVariable:any[] = ["Juan", 25, true];
listaVariable[0] = 50;
```

Definición de tipos propios (type)

persona.ts

```ts
export type Persona={
    nombre:string,
    apellido:string,
    edad?:number
}
```

Atributo opcional

personas.ts

```ts
import {Persona} from "./persona";

let persona1:Persona={
    nombre:"Juan",
    apellido:"Perez",
    edad:23
}

let persona2:Persona={
    nombre:"Julian",
    apellido:"Perez",
}

let persona3:Persona={
    nombre:"Fermín",
    apellido:"Perez",
    edad:"Hola"
}
```

Error

Definición de interfaces

- Una interfaz contiene la definición de propiedades y métodos de un objeto, pero no su implementación

```ts
export interface Persona {
    nombre:string;
    apellido:string;
    edad?:number;
}

let persona1:Persona={
    nombre:"Juan",
    apellido:"Perez",
    edad:23
}
```

```ts
interface Charlante {
    nombre:string;
    apellido:string;
    edad?:number;
    saluda():string;
}
```

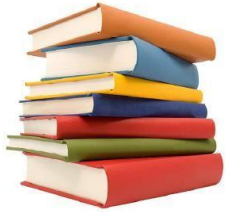
```ts
let charlante1:Charlante={
    nombre:"Juan",
    apellido:"Perez",
    edad:23
}

let charlante2:Charlante={
    nombre:"Juan",
    apellido:"Perez",
    edad:23,
    saluda:()=>{
        return "Hola";
    }
}
```

Interfaces vs tipos

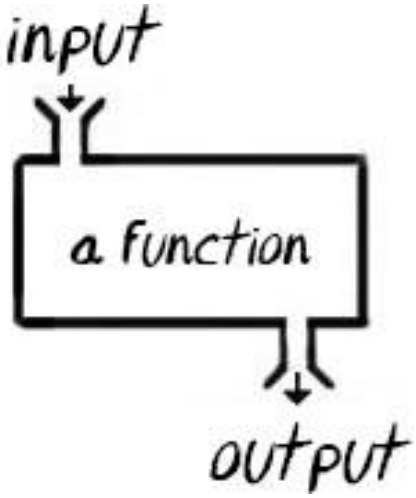
- Son dos maneras muy similares de definir nuestros tipos
- Dependen de las convenciones usadas por el equipo de trabajo
- Habitualmente:
  - Los tipos se utilizan más cuando usamos programación funcional.
  - Las interfaces habitualmente se usan cuando usamos Programación Orientada a Objetos

Ejemplo 1

- Vamos a reescribir el ejercicio de la biblioteca (parte 1) usando tipos/interfaces
  - Comenta las partes relativas a funciones
  - Define un nuevo tipo/interfaz llamado `Libro` en un archivo externo



Compila el JS a la carpeta ./js



Funciones y objetos

Indice

Definición de funciones

Tipo de los parámetros

Tipo devuelto

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

Valor por defecto

Parametro opcional

```ts
function construirNombre(nombre: string=´Javi´, apellido?: string):
string {
    if(apellido)
        return `${nombre} ${apellido}`;
    else
        return `${nombre}`;
}
```

Tipo devuelto por una función

- Especificamos **`void`** si la función no devuelve nada

```ts
function imprimeError(mensaje: string): void {
    console.log("Error: " + mensaje);
}
```

- Especificamos **`never`** si la función va a devolver una excepción

```ts
function devuelveError(mensaje: string): never {
    throw new Error(mensaje);
}
```

Tipo devuelto por una función

- Podemos indicar que hay varias opciones

```
function compruebaNombre(mensaje: string): void | boolean {
    if(nombre== "Juan")
        return true;
    else
        console.log("Error: " + mensaje);
}
```

Funciones flecha

JS

```
const imprimeNombre=(nombre)=>
        console.log("Me llamo ${nombre}");
```

TS

```
const imprimeNombre=(nombre:string):void=>
        console.log("Me llamo ${nombre}");
```

Funciones y parámetros desestructurados

JS

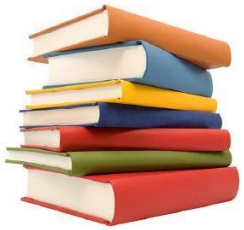
```
function imprimeNombre({nombre}) {
        console.log("Me llamo ${nombre}");
}
```

TS

```
function imprimeNombre({nombre}:{nombre:string}):void {
        console.log("Me llamo ${nombre}");
}
```

Ejemplo 1

- Vamos a reescribir el ejercicio de la biblioteca (parte 2) usando Typescript
  - Decomenta las partes relativas a funciones y añade los tipos correspondientes en las funciones (signaturas)



Compila el JS a la carpeta ./js

Ejemplo 2

- Vamos a reescribir una versión del “Piedra Papel o Tijera” usando interfaces en Typescript
  - Define un tipo para la jugada
  - Define la signatura (tipos) de las funciones

Compila el JS a la carpeta ./js

Definición de clases

```ts
class  Persona {
    private  nombre: string;
    private  edad: number;
    private  direccion: string;
    constructor(nombre: string, edad: number) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public  saludar(): void {
        console.log(`Hola, mi nombre es ${  this.nombre  } y tengo ${
                        this.edad  } años.`);
    }
    public  definirDireccion(direccion: string): void {
        this.direccion = direccion;
    }
}

let  persona = new  Persona("Javier Gonzalez", 32);
persona.saludar();
persona.definirDireccion("Calle Falsa, 123");
```

Visibilidad de atributos

- Podemos modificar visibilidad con los atributos
  - Pública (`public`)
    - El atributo es accesible desde fuera de la clase
  - Privada (`private`)
    - El atributo solo es accesible desde dentro de la clase
    - Debería definir ***getters*/*setters*** para el acceso al mismo
  - Protegida (`protected`)
    - Los atributos solo son accesibles desde dentro de la clase y las clases que heredan de la clase actual

```
Por defecto los atributos son públicos
```

Elementos estáticos y readonly

- Elementos estáticos (`static`).
  - Propiedades / métodos que pertenecen a la clase, no a los objetos
- Atributos de solo lectura (`readonly`)
  - Atributos que solo se pueden leer y no se pueden modificar
  - Se inicializa al declararla o en el constructor

Herencia

Herencia de clase mediante extends

```ts
class Person {
    public name: string = 'Javi';
    public saluda() {
      console.log(`Hola ${this.name}!`);
    }
  }

  class Developer extends Person {
    public languages: string[] = ['Java', 'JavaScript'];
  }

  const developer = new Developer();

  console.log(developer.name); // 'Javi'
  developer.saluda(); // Hi Javi!
```

Herencia usando interfaces

Herencia de interface mediante `implements`

```
interface Adulto {
    esAdulto(): boolean;
  }

  class Persona implements Adulto {
    protected name: string = 'Javi';
    private age: number = 28;

    esAdulto() {
      return this.age >= 18;
    }
  }

  const persona = new Persona();

  console.log(persona.esAdulto()); // true
'
  developer.saluda(); // Hi Javi!
```

Herencia múltiple

```ts
interface Adulto {
    esAdulto(): boolean;
  }

  interface Charlatan {
    habla(): void;
  }

  class Persona implements Adulto,Charlatan {
    protected name: string = 'Javi';
    private age: number = 28;

    esAdulto() {
      return this.age >= 18;
    }
    habla(){
        return "Bla bla bla";
    }
  }

  const persona = new Persona();

  console.log(persona.esAdulto()); // true
  console.log(persona.habla()); // Bla bla bla
```

Clases abstractas

- Son clases de las cuales no voy a definir ninguna instancia
- Tienen métodos abstractos (sin definir)
  - Pero podemos tener otros métodos definidos
- **Una clase que tiene todos sus métodos abstractos es una interfaz (`interface`).**

Se usan cuando queremos definir una funcionalidad común a un conjunto de objectos

Usando clases abstractas

```
abstract class Figura{
    protected color:string;
    public constructor(color:string){
        this.color=color;
    }
    public abstract dibuja():void;
}
```

```
class Circulo extends Figura{
    public dibuja():void{
        console.log(`Soy un circulo ${this.color}`);
    }
}

class Cuadrado extends Figura{
    public dibuja():void{
        console.log(`Soy un cuadrado ${this.color}`)
    }
}
```

```
let figuras:Figura[]=new Array(2)

figuras[0]=new Circulo("rojo");
figuras[0].dibuja();

figuras[1]=new Cuadrado("azul");
figuras[1].dibuja();
```

Ejemplo 3

- Vamos a reescribir la familia usando interfaces y herencia en Typescript
  - Reescribe las propiedades
  - Establece las propiedades privadas y los métodos públicos
  - Establece los parámetros opcionales

Compila el JS a la carpeta ./js