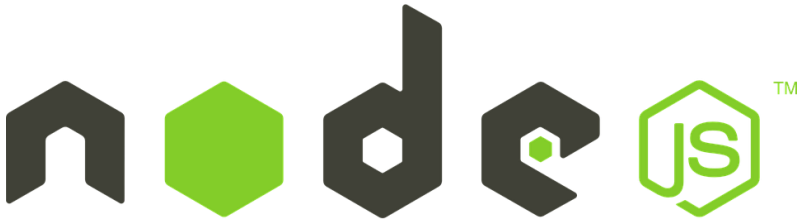
DESARROLLO DE APLICACIONES CON ANGULAR

TypeScript

TYPESCRIPT

- Node.js
  - Entorno de ejecución de JavaScript, construido sobre el motor V8.
  - Motor V8:
    - Desarrollado por Google.
    - Motor de JavaScript.
    - Implementa ECMAScript y WebAssembly.
      - ECMAScript. Especificación de un lenguaje de programación. JavaScript es una implementación de ECMAScript.
      - WebAssembly. Formato de código binario ejecutable en los navegadores.

TYPESCRIPT



- Node.js
  - https://nodejs.org/
  - Instalación versión LTS
    - Instalar las herramientas, incluido Chocolatey (gestor de paquetes).
  - Evaluación de versiones:
    - node --version (Node)
    - npm --version (instalador de paquetes de Node)

TYPESCRIPT

- Typescript
  - https://www.typescriptlang.org/
  - Instalación:
    - npm install -g typescript
  - "Compilación" de TypeScript:
    - tsc *fichero.ts* → Genera *fichero.js*
  - Ejecución de JavaScript:
    - node *fichero.js*

TYPESCRIPT

- Typescript. Configuración de la "compilación":
  - Creando un fichero `tsconfig.json` en la carpeta del proyecto.
  - Compilando sin especificar ficheros (ejecutando tsc).

```
{
    "compilerOptions": {
        "target":"es2021",
        "outDir":"./build",
        "noEmitOnError":true
    }
}
```

https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

TYPESCRIPT

- TypeScript. Variables.
  - Declaración de variables:
    - let *nombre_variable*:*tipo*
    - Se puede declarar una variable sin indicar tipo, realizándose una inferencia de este en función de la inicialización.
    - Si no se inicializa, se infiere el tipo **`any`**.
  - Tipos:
    - Primitivos: string, number y boolean.
    - Arrays → `nombre_array = tipo[]`
    - Arrays → `let nombre_array=Array<tipo>()`

https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

TYPESCRIPT

- TypeScript. Variables.
  - Tipos:
    - any → Admite cualquier valor (tipo genérico).
      - TypeScript infiere el tipo any de forma implícita si no se indica tipo. Esto funcionará por defecto, salvo que se le indique al compilador la opción `"noImplicitAny":true`

```
function funcion(parametro){


}
```

TYPESCRIPT

- TypeScript. Variables.
  - Tipos:
    - Object Type → Permite crear tipos compuestos.

```
let punto:{x:number, y:number};
punto={x:10,y:20};
console.log("(Punto):x:" + punto.x + ":y:" + punto.y);
```

TYPESCRIPT

- TypeScript. Variables.
  - Tipos:
    - Union Types → Permite crear tipos alternativos (la variable puede adoptar un valor compatible con cualquiera de los tipos).
    - Los tipos se separan por el símbolo "|".

```
let combinado: string | number | boolean;
combinado = "Texto";
combinado = 8;
combinado = true;
```

TYPESCRIPT

- TypeScript. Variables.
  - Tipos:
    - Type Aliases→ Permite crear nuevos tipos y asignarles un nombre.
    - Permiten la reutilización del tipo.

```
type direccion = string;
let miDireccion:direccion = "Cáceres";


type punto = {x:number, y:number};
let posicion:punto = {x:10, y:5};
```

TYPESCRIPT

- TypeScript. Arrays.
  - Recorrer array:
    - for (let variable of array)
  - Métodos:
    - pop. Elimina el último elemento.
    - push. Agrega nuevo elemento.
    - splice. Elimina elementos de un array.
    - slice: Genera una copia de una array en la que se eliminan elementos.
    - filter: Genera una copia del array con los elementos que pasan el filtro.

TYPESCRIPT

- TypeScript. Variables.
  - Valores null y undefined:
    - Ambos valores se pueden asignar a una variable de cualquier tipo.
    - undefined significa que no se le ha asignado valor aún.
    - null significa que no se le ha querido asignar valor.
    - La comparación devuelve true.
    - La comparación estricta devuelve false.

```
let nulo:number = null;
let indefinido:string = undefined;
```

TYPESCRIPT

- TypeScript. Funciones.
  - function *nombre(parámetro:tipo, parámetro:tipo):tipo*

```
function sumar(s1:number, s2:number):number {
    return s1 + s2
}
let resultado = sumar(3,8);
console.log(resultado);
```

TYPESCRIPT

- TypeScript. Funciones anónimas.
  - Sólo sirven para un único uso.
  - No necesitan la declaración de la función.

```
let laborables =
["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];

laborables.forEach(function(dia){
    console.log(dia);
});
```

TYPESCRIPT

- TypeScript. Funciones "arrow".
  - Sólo sirven para un único uso.
  - Son una versión compacta de las funciones anónimas.

```
let laborables =
["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];

laborables.forEach((dia)=> {
    console.log(dia);
});
```

MÓDULO 1: TYPESCRIPT

- TypeScript. Funciones con parámetros de tipo Object Type.

```
function calcular(punto: {x:number,
y:number}){
    let resultado:number = punto.x +
punto.y
    return resultado;
}

let calculo = calcular({x:5,y:7});
console.log(calculo);
```

TYPESCRIPT

- TypeScript. Funciones con parámetros opcionales.
  - Se indica con el símbolo "?" después del nombre del parámetro.

```
function traducir(nombre? : string){
    if (nombre){
        console.log("Traduciendo " + nombre);
    } else {
        console.log("No hay nombre");
    }
}

traducir();
traducir("Palabra");
```

TYPESCRIPT

- TypeScript. Funciones con parámetros opcionales.
  - Pueden existir varios parámetros opcionales pero estos siempre deben estar en las últimas posiciones.
    - (OK) function traducir(nombre : string, apellido?: string)
    - (OK) function traducir(nombre? : string, apellido?: string)
    - (KO) function traducir(nombre? : string, **apellido**: string)
    - (OK) function calcular(punto: {x:number, y?:number})

TYPESCRIPT

- TypeScript. Interfaces.
  - Permiten definir nuevos tipos de datos (alternativa a Object Type).

```
interface Punto {
    x: number;
    y: number;
}

let p1:Punto;
p1.x=5;
p1.y=3;
```

TYPESCRIPT

- TypeScript. Interfaces.
  - La diferencia con Type Aliases es que las interfaces se pueden heredar y ampliar.

```
interface Punto {
    x: number;
    y: number;
    offset: number;//Ampliación
}

interface Punto3D extends Punto {
    z: number;//Herencia
}

let puntoInicial:Punto3D;
puntoInicial.x = 10;
puntoInicial.y = 20;
puntoInicial.z = 30;
puntoInicial.offset = -2;
```

TYPESCRIPT

- TypeScript. Constantes y literales.
  - Permiten asignar valores inmutables.

```
const numeroDias = 7;//Constantes
let numeroSemanas:8 = 8;//Literal
let nombreDia:"lunes"="lunes";//Literal
numeroDias=6;//ERROR
numeroSemanas=6;//ERROR
nombreDia="martes";//ERROR
```

TYPESCRIPT

- TypeScript. Constantes y literales.
  - Template literal.
    - Permite construir cadenas como consecuencia de la concatenación de literales y variables mediante expresiones.
    - Utiliza el símbolo de la comilla de ejecución o invertida `
    - Ejemplo:

```
console.log(`Texto del literal ${variable} continua texto
${objeto.atributo}`);
```

TYPESCRIPT

- TypeScript. Enumeraciones.
  - Permiten crear tipos con valores acotados.

```
enum Plataformas {
    PlayStation5,
    PlayStation4,
    XboxSeries,
    XboxOne,
    NintendoSwitch
}

let plataforma:Plataformas;
plataforma=Plataformas.PlayStation4;
```

https://www.typescriptlang.org/docs/handbook/enums.html

TYPESCRIPT

- TypeScript. Enumeraciones.
  - Se pueden asignar valores a las distintas opciones.

```
enum Plataformas {
    PlayStation5="PS5",
    PlayStation4="PS4",
    XboxSeries="XBS",
    XboxOne="XBO",
    NintendoSwitch="NS"
}
if ("PS4"==Plataformas.PlayStation4){
    console.log("Correcto");
}
```

TYPESCRIPT

- TypeScript. Clases.

```
class Factura {
    numero:number;
    cliente:string;
    constructor(numero:number, cliente:string){
        this.numero = numero;
        this.cliente = cliente;
    }
}
```

TYPESCRIPT

- TypeScript. Atributos de sólo lectura.

```
class Factura {
    readonly numero:number;
    cliente:string;
}

let f1 = new Factura();
f1.numero = 1;//ERROR
f1.cliente = "Refrescos S.L.";
```

TYPESCRIPT

- TypeScript. Inicialización de atributos.

```
class Factura {
    numero:number;
    cliente:string;
    importe:number = 50;//Inicialización
}
```

TYPESCRIPT

- TypeScript. Valores por defecto en los constructores.

```
class Factura {
    numero:number;
    cliente:string;
    constructor(numero:number, cliente:string="Desconocido"){
        this.numero = numero;
        this.cliente = cliente;
    }
}
```

TYPESCRIPT

- TypeScript. Sobrecarga de constructores. Sólo se admite un constructor implementado.

```
class Factura {
    numero:number;
    cliente:string;
    constructor(numero:number, cliente?:string){
        this.numero = numero;
        this.cliente = cliente;
    }
}
let f1 = new Factura(10);
let f2 = new Factura(10, "Editorial");
```

TYPESCRIPT

- TypeScript. Sobrecarga de constructores. Sólo se admite un constructor implementado.

```
class Factura {
    numero:number;
    cliente:string;
    constructor(numero:number);
    constructor(numero:number, cliente:string);
    constructor(numero:number, cliente?:string){
        this.numero = numero;
        this.cliente = cliente;
    }
}
let f1 = new
let f2 = new
new Factura()
```

```
2/2  Factura(numero: number, cliente: string): Factura
```

TYPESCRIPT

- TypeScript. Herencia

```
class FacturaDetallada extends Factura {
    constructor(){
        super(10,"El cliente");
    }
}
```

TYPESCRIPT

- TypeScript. Métodos

```
class Factura {
    numero:number;
    cliente:string;
    constructor(numero:number, cliente?:string){
        this.numero = numero;
        this.cliente = cliente;
    }
    facturar(){
        console.log("Generando factura...");
    }
}
```

TYPESCRIPT

- TypeScript. Atributos privados, "getter" y "setter"

```
class Factura {
    numero:number;
    private _cliente:string;
    constructor(numero:number, cliente?:string){
        this.numero = numero;
        this._cliente = cliente;
    }
    get cliente(){
        return this._cliente;
    }
    set cliente(valor){
        this._cliente = valor;
    }
```

TYPESCRIPT

- TypeScript. Visibilidad de atributos y métodos.
  - public
  - private
  - protected

```
class Clase {
    public atributo1:number;
    protected atributo2:number;
    private atributo3:number;
    public metodo1(){
        console.log("Ejecutando método 1...");
    }
    protected metodo2(){
        console.log("Ejecutando método 2...");
    }
    private metodo3(){
        console.log("Ejecutando método 3...");
    }
}
```

TYPESCRIPT

- TypeScript. Estáticos.
  - Se pueden utilizar tanto en atributos como en métodos.

```
class Clase {
    public static valorMaximo:number=15;
    public static saludar(){
        console.log("Saludando...");
    }
}
console.log(Clase.valorMaximo);
Clase.saludar();
```

TYPESCRIPT

- TypeScript. Clases abstractas.
  - Compuestas de métodos abstractos y concretos.

```
abstract class Clase {
    abstract metodoAbstracto():void;
    public metodoConcreto():void{
        console.log("Ejecutando método concreto...");
    }
}
class ClaseDerivada extends Clase {
    metodoAbstracto(): void {
        throw new Error("Method not implemented.");
    }
}
```

TYPESCRIPT

- Rest parameter.
  - Permite incluir un parámetro en una función con un número indeterminado de parámetros.
  - La recogida se realiza como un array.
  - El parámetro debe ser el último.

```
function calcular(...parametros:string[]):void{
    for (let p of parametros) {
        console.log(p);
    }
}
calcular("uno","dos","tres");
```

TYPESCRIPT

- Spread operator.
  - Permite crear una copia de un array o de parte de un array en otro array.
  - También se puede aplicar a objetos.

```
let array=[1,2,3]
let copia = array;
array[0]=8;
console.log(copia[0]);
```

Copia referencia

```
let array=[1,2,3]
let copia = [...array];
array[0]=8;
console.log(copia[0]);
```

Copia valores

TYPESCRIPT

- setTimeout y setInterval
  - Métodos asíncronos.
  - setTimeout. Permite demorar la ejecución de un bloque de código.
  - setInterval. Permite establecer un ciclo de ejecución