FULL STACK DEVELOPER

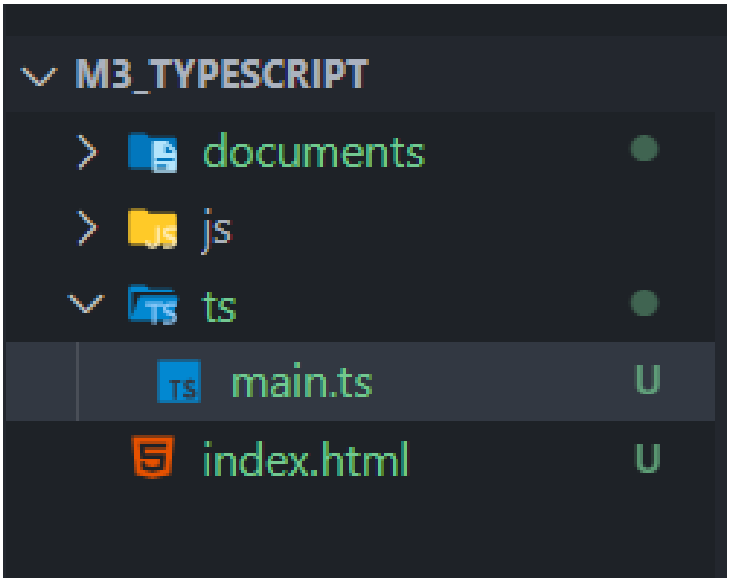
Módulo 3 – TypeScript

Jorge Borrego Marqués

Prueba Evaluable

1. Crea una estructura de proyecto HTML y TypeScript/JavaScript.

- Folder de Proyecto .\proyectos\M3_TypeScript, añadimos en la raíz un index.htm(enlazando un script con el archivo main.js transpilado)l, un folder ts con un archivo main.ts (dónde desarrollaremos el código del TypeScript y un folder vacío js donde se alojará el main.js que resultará de transpilar el TypeScript en JavaScript para poder ser interpretado por el navegador.



2. En el archivo index.html, usa el siguiente código para crear un sencillo formulario.

- Index.html

En el archivo index.html pego el código del formulario propuesto en la práctica y añadimos un script con la ruta del main.js (archivo de JavaScript) que interpretará el navegador.

index.html U × main.ts U

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Práctica Evaluable TypeScript</title>
  </head>
  <body>
    <label for="name">
      Nombre
      <input type="text" id="name" />
    </label>
    <label for="surname">
      Apellidos
      <input type="text" id="surname" />
    </label>
    <br />
    <label for="email">
      Correo electrónico
      <input type="email" id="email" />
    </label>
    <label for="birthDate">
      Fecha de nacimiento
      <input type="date" id="birthDate" />
    </label>
    <br />
    <label for="sdUnit">
      Unidad de venta
      <input type="text" id="sdUnit" />
    </label>
    <label for="area">
      Zona geográfica
      <input type="text" id="area" />
    </label>
    <br />
    <button onclick="addEmployee()">Aceptar</button>
    <script src="./js/main.js"></script>
  </body>
</html>
```

3. Desarrolla una interfaz TypeScript para empleados que contenga las propiedades de nombre, apellidos, correo electrónico y fecha de nacimiento con sus correspondientes tipos.

Interfaz Employee con los miembros requeridos name, surname, email y birthDate.

- Name : Es un tipo primitivo ‘string’
- Surname: Es un tipo primitivo ‘string’
- Email: Es un tipo primitivo ‘string’
- birthDate: Es un tipo object de la clase Date.

La propiedad “birthDate” tiene un tipo “object” ya que se obtendrá de forma global, a partir, de la clase Date. De esta forma debemos instanciar un objeto de la clase global y a partir de dicha instancia poder usar sus propios métodos.

```
interface Employee {
    name        :   string;
    surname     :   string;
    email       :   string;
    birthDate   :   Date;
}
```

4. Desarrolla una clase TypeScript que implemente la interfaz anterior y además contenga las propiedades específicas para empleados de ventas “Unidad de venta” y “Zona geográfica”.

Para definir los tipos de las propiedades de la clase “SaleEmployee” usamos la palabra “implements” que, como su propio nombre indica, implementa los tipos definidos en la interfaz “Employee” e impide que no puedan tomar otros tipos diferentes en el futuro desarrollo de la aplicación.

Una vez definida la clase e implementada la interfaz, definimos las propiedades correspondientes, y añadimos un par de propiedades nuevas:

- sdUnit: de tipo primitivo string
- area: de tipo primitivo string.

```
class SalesEmployee implements Employee {

    name        :   string;
    surname     :   string;
    email       :   string;
    birthDate   :   Date;
    sdUnit      :   string;
    area        :   string;
}
```

5. Añade a la clase anterior el método constructor y los métodos "get" y "set" de las propiedades

```
// Método Constructor
    constructor(name: string, surname: string,
email: string, birthDate: string, sdUnit: string,
area: string) {
        this.name      =   name;
        this.surname   =   surname;
        this.email     =   email;
        this.birthDate =   new Date(birthDate);
        this.sdUnit    =   sdUnit;
        this.area      =   area;
    }
    // Métodos Setter y Getters
    public setName(v : string) {
        this.name  =   v;
    }
    public getName() : string {
        return this.name;
    }
    public setSurname(v : string) {
        this.surname  =   v;
    }
    public getSurname() : string {
        return this.surname;
    }
    public setEmail(v : string) {
        this.email  =   v;
    }
    public getEmail() : string {
        return this.email;
    }
    public setBirthDate(v: string){
```

```
        this.birthDate = new Date(v);
    }
    public getBirthDate(): Date {
        return this.birthDate;
    }
    public setSdUnit(v : string) {
        this.sdUnit  =   v;
    }
    public getSdUnit() : string {
        return this.sdUnit;
    }
    public setArea(v : string) {
        this.area  =   v;
    }
    public getArea() : string {
        return this.area;
    }
```

Definimos el método constructor asignando a cada propiedad, parámetros compatibles con los tipos implementados en la interface “Employee”.

Es importante resaltar que la propiedad “birthDate” va a recibir en el método constructor una parámetro del tipo “string”. No podemos asignar un parámetro de un tipo (en este caso “string”) a una propiedad con un tipo diferente (en este caso concreto “Date”), por lo que debemos usar el parámetro para crear una instancia de la clase global “new Date(birthDate:string)”.

De la misma forma que usamos este procedimiento en el método constructor, debemos usarlo también en el método “getBirthDate(): Date” De esta forma podremos usar los métodos propios de la clase.

6. Declara la función que es invocada desde el botón index.html con el nombre “addEmployee()” para que instancie un objeto de la clase “empleado de ventas” declarada anteriormente, inicializándolo con los valores del formulario y, en la misma función, emplea los métodos de la clase para imprimir por consola sus valores.

En este punto vamos a poner en práctica lo aprendido en el módulo anterior de ECMAScript2015. Definiremos una función con un protocolo async-await para recoger los datos del formulario y validarlos por medio de una función Promise. La función se resolverá con una instancia del objeto de la clase definida.

Función async-await que recoge los datos del formulario y los imprime por consola usando los métodos de clase.

```
// * Función async-await *
async function addEmployee() {
    try {
        let salesEmployee = await setEmployee(

(<HTMLInputElement>document.getElementById('name')).v
alue.toLocaleUpperCase(),

(<HTMLInputElement>document.getElementById('surname')
).value.toLocaleUpperCase(),

(<HTMLInputElement>document.getElementById('email')).
value.toLocaleLowerCase(),
```

```
(<HTMLInputElement>document.getElementById('birthDate
')).value,

(<HTMLInputElement>document.getElementById('sdUnit'))
.value.toLocaleUpperCase(),

(<HTMLInputElement>document.getElementById('area')).v
alue.toLocaleUpperCase()
        );
        console.log(`
            Name:       ${salesEmployee.getName()},
            Surname:
 ${salesEmployee.getSurname()},
            Email:      ${salesEmployee.getEmail()},
            Birthday:
                Day:
 ${salesEmployee.getBirthDate().getDate()}
                Month:
 ${salesEmployee.getBirthDate().getMonth()}
                Year:
${salesEmployee.getBirthDate().getFullYear()}
            Department: ${salesEmployee.getSdUnit()},
            Area:       ${salesEmployee.getArea()}

            `);
    } catch (error) {
        console.error(error);
    }
}
```

Implementamos la función con un bloque “try-catch” imprimiendo por consola el posible error. Recoge los valores de los campos directamente como argumentos de una función y pasados por un “toLocalUpperCase()”, sólo con motivos estéticos, para mostrarlos por consola.

Función Promise que nos sirve para validar campos vacíos y resuelve con instancia de Clase.

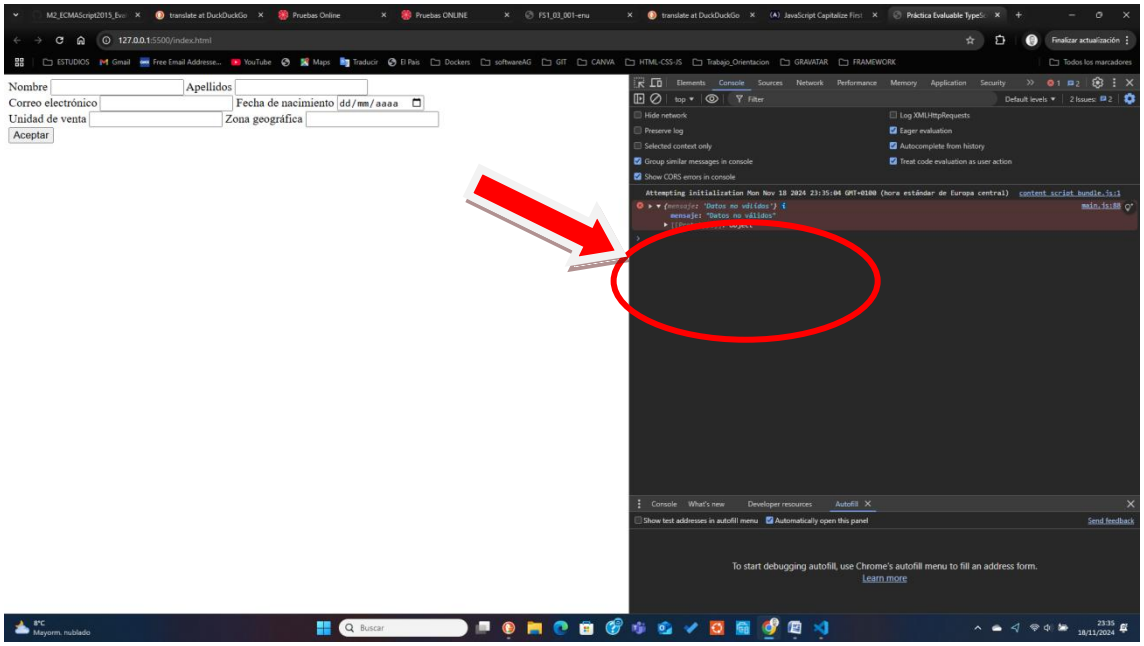
```
// * Función Promise *
function setEmployee(name:string, surname: string,
email: string, birthDate: string, sdUnit: string,
area: string): any {
    return new Promise<SalesEmployee>((resolve,
reject) => {
        if (
            name === '' || surname === '' || email
=== '' || birthDate === '' || sdUnit === '' || area
=== ''
        )
        reject({mensaje: `Datos no válidos`});
        setTimeout(() => {
            resolve(new
SalesEmployee(name,surname,email,birthDate,sdUnit,are
a))
        },2000);
    })
}
```

Tipamos el retorno de la function con “: any” , ya que puede devolvernos un “string” por el método reject de la Promesa o en el caso de resolverse, devuelve una instancia de la clase <SalesEmployee>. Simulamos un retardo con el método setTimeout() de 2 sg.

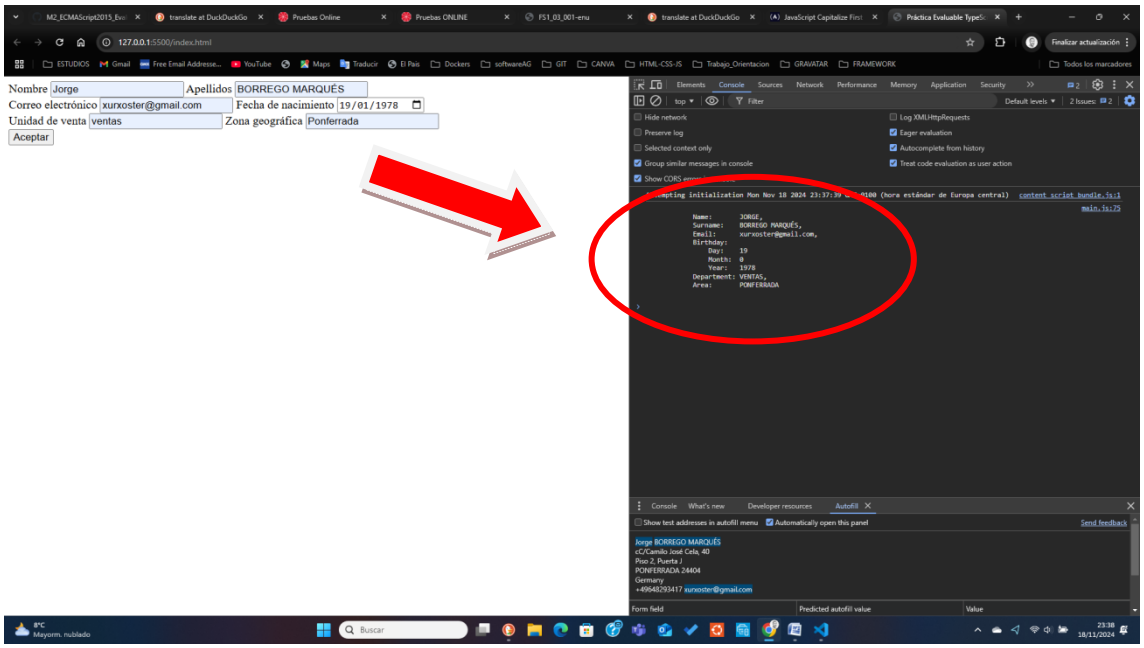
7. Transpila el código a JavaScript y comprueba su funcionamiento en la consola de las herramientas de desarrollador.

Respuesta por Consola:

- Campos vacíos



- Campos rellenos OK.



8. Código completo de la tarea.(ts/main.ts)

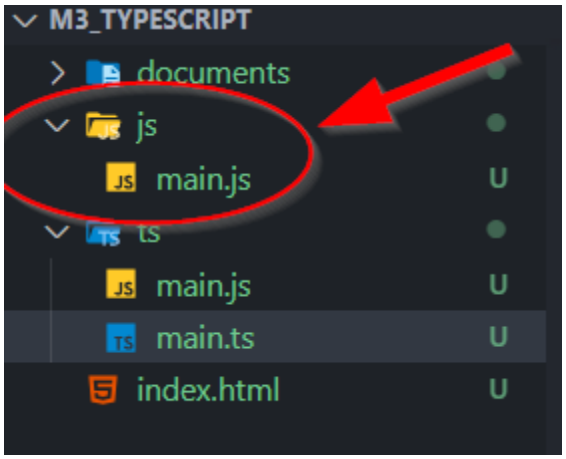
```typescript
// ** Prueba Evaluable TypeScript **

// * Interface *

interface Employee {
    name        :   string;
    surname     :   string;
    email       :   string;
    birthDate   :   Date;
}


// * Clase Empleado de Ventas *
class SalesEmployee implements Employee {

    // Definición de Propiedades
    name        :   string;
    surname     :   string;
    email       :   string;
    birthDate   :   Date;
    sdUnit      :   string;
    area        :   string;

    // Método Constructor
    constructor(name: string, surname: string,
email: string, birthDate: string, sdUnit: string,
area: string) {
        this.name      =   name;
        this.surname   =   surname;
        this.email     =   email;
        this.birthDate =   new Date(birthDate);
        this.sdUnit    =   sdUnit;
        this.area      =   area;
```

```
    }
    // Métodos Setter y Getters
    public setName(v : string) {
        this.name  =   v;
    }
    public getName() : string {
        return this.name;
    }
    public setSurname(v : string) {
        this.surname  =   v;
    }
    public getSurname() : string {
        return this.surname;
    }
    public setEmail(v : string) {
        this.email  =   v;
    }
    public getEmail() : string {
        return this.email;
    }
    public setBirthDate(v: string){
        this.birthDate = new Date(v);
    }
    public getBirthDate(): Date {
        return this.birthDate;
    }
    public setSdUnit(v : string) {
        this.sdUnit  =   v;
    }
    public getSdUnit() : string {
        return this.sdUnit;
    }
    public setArea(v : string) {
        this.area  =   v;
    }
```

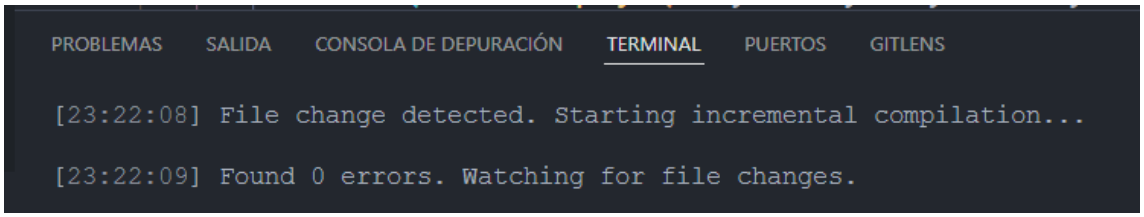
```
    public getArea() : string {
        return this.area;
    }
}

// * Función Promise *
function setEmployee(name:string, surname: string,
email: string, birthDate: string, sdUnit: string,
area: string): any {
    return new Promise<SalesEmployee>((resolve,
reject) => {
        if (
            name === '' || surname === '' || email
=== '' || birthDate === '' || sdUnit === '' || area
=== ''
        )
        reject({mensaje: `Datos no válidos`});
        setTimeout(() => {
            resolve(new
SalesEmployee(name,surname,email,birthDate,sdUnit,a
rea))
        },2000);
    })
}

// * Función async-await *
async function addEmployee() {
    try {
        let salesEmployee = await setEmployee(

(<HTMLInputElement>document.getElementById('name'))
.value.toLocaleUpperCase(),

(<HTMLInputElement>document.getElementById('surname
')).value.toLocaleUpperCase(),
```

```
(<HTMLInputElement>document.getElementById('email')
).value.toLocaleLowerCase(),

(<HTMLInputElement>document.getElementById('birthDa
te')).value,

(<HTMLInputElement>document.getElementById('sdUnit'
)).value.toLocaleUpperCase(),

(<HTMLInputElement>document.getElementById('area'))
.value.toLocaleUpperCase()
        );
        console.log(`
            Name:       ${salesEmployee.getName()},
            Surname:
 ${salesEmployee.getSurname()},
            Email:
 ${salesEmployee.getEmail()},
            Birthday:
                Day:
 ${salesEmployee.getBirthDate().getDate()}
                Month:
 ${salesEmployee.getBirthDate().getMonth()}
                Year:
${salesEmployee.getBirthDate().getFullYear()}
            Department:
${salesEmployee.getSdUnit()},
            Area:       ${salesEmployee.getArea()}

            `);
    } catch (error) {
        console.error(error);
    }
}
```

9. Código transpilado OK.(js/main.js)



Transpilado sin errores:



Código transpilado JavaScript:

```
var __awaiter = (this && this.__awaiter) || function
(thisArg, _arguments, P, generator) {
    function adopt(value) { return value instanceof P
? value : new P(function (resolve) { resolve(value);
}); }
    return new (P || (P = Promise))(function
(resolve, reject) {
        function fulfilled(value) { try {
step(generator.next(value)); } catch (e) { reject(e);
} }
        function rejected(value) { try {
step(generator["throw"](value)); } catch (e) {
reject(e); } }
```

```
        function step(result) { result.done ?
resolve(result.value) :
adopt(result.value).then(fulfilled, rejected); }
        step((generator = generator.apply(thisArg,
_arguments || [])).next());
    });
};
// ** Prueba Evaluable TypeScript **
// * Clase Empleado de Ventas *
class SalesEmployee {
    // Método Constructor
    constructor(name, surname, email, birthDate,
sdUnit, area) {
        this.name = name;
        this.surname = surname;
        this.email = email;
        this.birthDate = new Date(birthDate);
        this.sdUnit = sdUnit;
        this.area = area;
    }
    // Métodos Setter y Getters
    setName(v) {
        this.name = v;
    }
    getName() {
        return this.name;
    }
    setSurname(v) {
        this.surname = v;
    }
    getSurname() {
        return this.surname;
    }
    setEmail(v) {
        this.email = v;
```

```
    }
    getEmail() {
        return this.email;
    }
    setBirthDate(v) {
        this.birthDate = new Date(v);
    }
    getBirthDate() {
        return this.birthDate;
    }
    setSdUnit(v) {
        this.sdUnit = v;
    }
    getSdUnit() {
        return this.sdUnit;
    }
    setArea(v) {
        this.area = v;
    }
    getArea() {
        return this.area;
    }
}
// * Función Promise *
function setEmployee(name, surname, email, birthDate,
sdUnit, area) {
    return new Promise((resolve, reject) => {
        if (name === '' || surname === '' || email
=== '' || birthDate === '' || sdUnit === '' || area
=== '')
            reject({ mensaje: `Datos no válidos` });
        setTimeout(() => {
            resolve(new SalesEmployee(name, surname,
email, birthDate, sdUnit, area));
        }, 2000);
```

```
    });
}
// * Función async-await *
function addEmployee() {
    return __awaiter(this, void 0, void 0, function*
() {
        try {
            let salesEmployee = yield
setEmployee(document.getElementById('name').value.toL
ocaleUpperCase(),
document.getElementById('surname').value.toLocaleUppe
rCase(),
document.getElementById('email').value.toLocaleLowerC
ase(), document.getElementById('birthDate').value,
document.getElementById('sdUnit').value.toLocaleUpper
Case(),
document.getElementById('area').value.toLocaleUpperCa
se());
            console.log(`
            Name:       ${salesEmployee.getName()},
            Surname:
 ${salesEmployee.getSurname()},
            Email:      ${salesEmployee.getEmail()},
            Birthday:
                Day:
 ${salesEmployee.getBirthDate().getDate()}
                Month:
 ${salesEmployee.getBirthDate().getMonth()}
                Year:
${salesEmployee.getBirthDate().getFullYear()}
            Department: ${salesEmployee.getSdUnit()},
            Area:       ${salesEmployee.getArea()}

            `);
        }
```

```
        catch (error) {
            console.error(error);
        }
    });
}
```