
TypeScript

TypeScript


GOBIERNO
DE ESPAÑA
MINISTERIO
DE INDUSTRIA, COMERCIO
Y TURISMO
EOI
Escuela de
organización
industrial


Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro


SISTEMA NACIONAL DE
GARANTÍA
JUVENIL

01 Introducción

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Historia

- Creado por Microsoft en 2012
  - Actualmente está en la versión 4.6 (Mayo 2022)
- Superconjunto de ECMAScript 6
  - Añade un fuerte tipado a JavaScript
- Compilador TS convierte código TS a código ES
  - Detecta errores en tiempo de compilación
  - Se puede especificar la versión de ES con la que se quiere trabajar

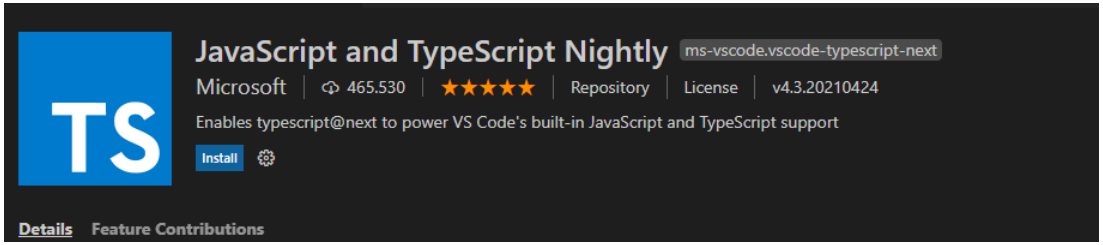
Configuración

- Requisitos
  - NodeJs y NPM
  - Paquete NodeJs ***typescript***

- Instalación
  - ***npm install –g typescript***
  - ***Instalando la extensión de Visual Studio Code***

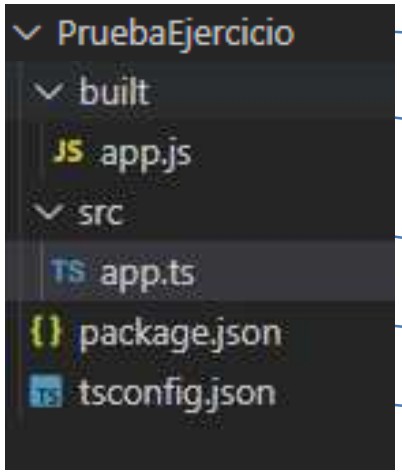
- Al usar la instalación con ***–g*** estamos indicando que el paquete va estar disponible de manera global en todo el sistema
- Podemos probar a ejecutar por línea de comandos ***tsc -v*** , esto nos mostrará la versión de ***TypeScript*** que tenemos instalada

```
[Typescript]:tsc -v
Version 4.2.4
```



Configuración : estructura carpeta ejercicios

- Estructura (Se recomienda usar el plugin Folder Templates)
  - Carpeta con ficheros ***.ts (src/)***
  - Carpeta con ficheros ***.js*** resultado de la compilación de los ficheros ***.ts (built/)***
  - Fichero de configuración de ***TypeScript*** para el proyecto (***tsconfig.json)***



Carpeta del ejercicio

Ficheros ***.js*** generados

Ficheros fuentes ***.ts***

Script para compilar todos los ficheros

Ficheros de configuración
TypeScript del proyecto ***ejercicio1***

Configuración : package.json

- Fichero de configuración de proyecto nodejs **package.*json***
  - Contiene los Script necesarios para compilar y ejecutar los ejercicios

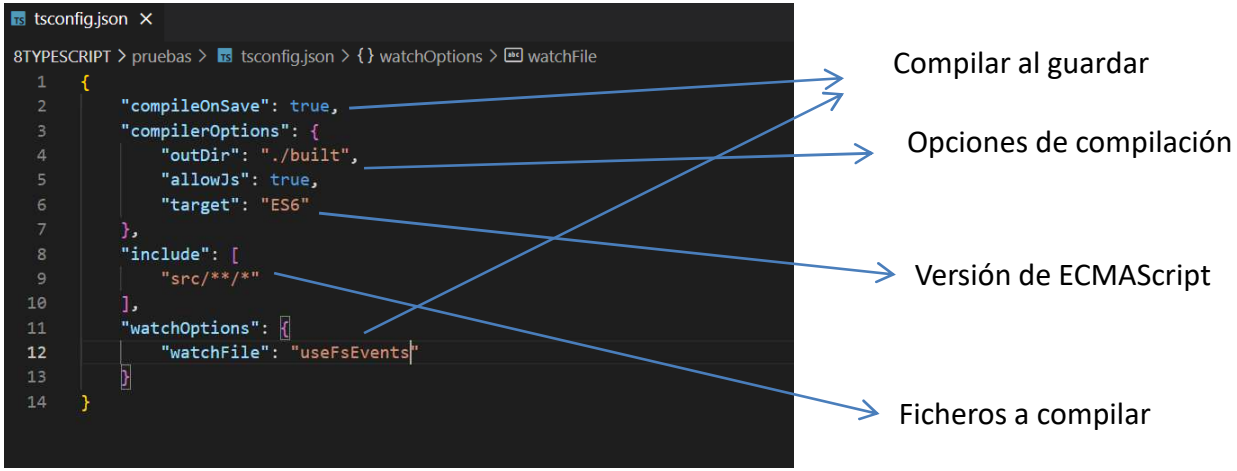
```
{
    "name": "pruebas_typescript",
    "version": "1.0.0",
    "description": "",
    "main": "app.js",
    ▷ Debug
    "scripts": {
        "compilar": "tsc -w",
        "ejecutar": "nodemon built/app.js"
    },
    "author": "EOI",
    "license": "ISC"
}
```

Compilación en modo Watch

Ejecución

Configuración : tsconfig.json

- Fichero de configuración ***tsconfig.json***
  - ➢ Contiene instrucciones de compilación y ejecución del proyecto, se ubica en el raíz del proyecto TypeScript



Compilando

- Siguiendo la estructura anterior y con el fichero de configuración ***tsconfig.json*** en el raíz del proyecto, tenemos dos opciones :
  - Ejecutar ***tsc*** sin argumentos en el directorio raíz del proyecto

    Al ejecutar ***tsc*** desde el raíz del proyecto , buscará el fichero de configuración ***tsconfig.json*** y compilará según la configuración del fichero de configuración.

    ```
    tsc
    ```

  - Ejecutar ***tsc –p [ruta a carpeta donde esté el fichero tsconfig.json]***

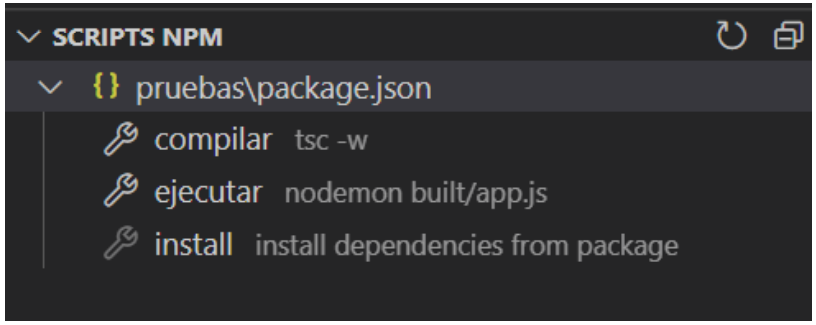
    ```
    tsc -p .
    ```

  - Compilar y ejecutar desde Visual Studio usando los Scripts definidos en package.json

Compilando

- En ambos casos, el resultado de la compilación del proyecto generará, si no ha habido errores de compilación, los ficheros ***.js*** Resultantes

- Para probar los ejemplos y ejercicios, primero ejecutaremos el script **compilar** y después el script **ejecutar**, a partir de este
- Momento cada vez que modifiquemos el código del fichero **app.ts** compilará y ejecutará automáticamente el proyecto



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

¿Qué genera la compilación?

- Tener muy presente que el código **TypeScript es compilado** y el **resultado de esta compilación es la generación de código JavaScript**

- **Podemos usar código JavaScript dentro de nuestro código TypeScript**

- Para poner comentarios en nuestro código, podemos hacerlo como lo hacíamos en JavaScript

- Podemos hacer un ejercicio para probar el uso de instrucciones JavaScript dentro del código TypeScript

Ejercicio1

- Generar una carpeta llamada ***Ejercicio1*** con la estructura mostrada anteriormente. Vamos a usar esta carpeta como base para el resto de ejercicios. Cuando hagamos un nuevo ejercicio , copiaremos esta carpeta y le cambiaremos el nombre por el del nuevo ejercicio.

Para este ejercicio en concreto, al ser el ejercicio base, mostrar por consola El mensaje ***"Hola compañeros de la EOI"***

02 Variables

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Tipos primitivos en Javascript

- **Number**
- **String**
- **BigInt**
- **Boolean**
- **Symbol**
- **Null**
- **Undefined**
- **Object**

GOBIERNO DE ESPAÑA MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Variables y constantes

- Las variables en TypeScript se declaran con la palabra clave ***let*** *o* ***const*** seguido del nombre de la variable, el carácter ***:*** y el tipo de variable . Las variables declaradas como **const** deben inicializarse siempre.

Ejemplos :

```
let edad:number;

let cierto: boolean;

let nombre:string;

let array: [];
```

- También podemos definir constantes con la palabra clave ***const***

```
const edad=21;
```

Tipos de datos

- ***boolean*** : como lo definíamos en JavaScript, puede tener dos valores Posibles (***true , false)***

```
let continua:boolean;
continua=true;
```

- **number** *:* todos los números en TypeScript son decimales , también se puede usar con valores hexadecimal, binario, octal ...

```
let edad:numer=21;
let altura:number=2.21;
```

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

Tipos de datos

- ***string***: cadenas de carácteres, como en JavaScript se permiten las dobles comillas o las comillas simples

let nombre:string='ANTONIO';

- **Array**: array de cualquier tipo de dato. Se puede definir de dos formas.
  1. *let <nombre_variable> : <tipo_dato> []*
  2. *let <nombre_variable> : Array<tipo_dato>;*

let edades:number[] = [21,30,45];

let edades:Array<number>=[21,30,45];

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

Tipos de datos

- ***Tuplas***: array con elementos en una posición determinada y fija y con tipo de datos también conocidos de antemano

```
let clienteEdad: [string,number];
clienteEdad =['ANTONIO',21];
console.log(clienteEdad[0])
```

- **enum**: conjunto de Enumerados. Asocia a cada clave un valor comenzando por el valor 0, ano ser que nosotros le indiquemos un valor inicial

```
enum Medallas {oro,plata,bronce};
let medalla: Medallas = Medallas.oro;

enum Medallas {oro=1,plata,bronce}
let medalla:Medallas=Medallas.oro
```

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

Tipos de datos

- ***any***: cualquier valor.

```
let desconocido: any;
desconocido='DESCONOCIDO';

desconocido=12;
```

- **Object***:* Objeto, como hemos visto en JavaScript o alguno definido por nosotros

```
let persona: Object={...}
```

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

Ejercicio2

- Probar los distintos tipos de datos vistos hasta ahora, asignarles valores y mostrarlos por consola.
  - ✓ boolean
  - ✓ number
  - ✓ string
  - ✓ array
  - ✓ tuplas
  - ✓ enum
  - ✓ any
  - ✓ Object

03

Operador Spread

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Tipos de datos

- **La sintaxis extendida o spread sintax (...)** permite copiar las propiedades de un Array u Objeto en otro Array u Objeto nuevo.
- Se procesan de izquierda a derecha, eso quiere decir que cualquier valor que venga después de procesar un ***Spread*** sobre escribirá el valor anterior
- Mejor lo vemos con ejemplos

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

Tipos de datos

- **Ejemplo de uso en Objetos:**

Si aplicamos

```
let menu= { menu: 'menu1', precio: '20'};
let nuevoMenu= { ...menu, menu: 'menu2' };
```

El resultado de nuevoMenu será

```
{ menu: 'menu2' ,precio:20};
```

Vemos como se ha copiado *menu* en *nuevoMenu,* pero al procesarse de Izquierda a derecho, el *menu:'menu2'* es el ultimo valor que se guarda

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

Tipos de datos

- **Ejemplo de uso en Arrays:**

Si aplicamos

```
let ca5= [ 0,2,3,4,5];
let ca7=[ ...ca,5,6,7]
```

El resultado de ca7 será

```
[ 0,2,3,4,5,6,7];
```

Si aplicamos

```
let ca5= [ 0,2,3,4,5,6,7];
let ca7=[ ...ca5,6,7]
```

El resultado de ca7 será

[0,2,3,4,5,6,7,6,7];

Ejercicio3

Implementar el ejemplo anterior y jugar con el operador ...

04 Módulos

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Agrupar funcionalidad por modulos

- Podríamos decir que un módulo es todo lo contenido en un fichero ***.ts***
- Podemos usar el contenido de un módulo en otro
- Podemos importar el código de otros ficheros ***.ts*** para usarlo dentro de nuestro fichero ***.ts***. Esto lo hacemos usando la palabra clave ***import***

import {<lo que queremos importar>} from '<ruta y fichero a importar sin extensión .ts>';

Para poder usar ***import*** antes deberemos hacer uso de la palabra clave ***export*** en el fichero que queramos importar. Solo se puede importar a un fichero ***.ts*** lo que se haya exportado desde otro. Todo lo que se exporte se tiene que realizar con el siguiente formato

export <lo que queramos exportar>;

Podemos exportar variables, clases, funciones , ...
Para poder usarlo hay que modificar la compilación añadiendo lo siguiente ***tsc –w --module commonjs*** en lugar de ***tsc*** como lo hacíamos hasta ahora

Agrupar funcionalidad por modulos

Ejemplo de uso de módulos :

*Fichero **usar.ts***

```
export  enum Medallas {oro=20,plata,bronce};
```

*Fichero **ejercicio.ts***

```
import {Medallas} from './usar';


let medalla:Medallas;
medalla=Medallas.oro;
console.log(medalla);
```

Ejercicio4

Implementar el ejemplo anterior para comprender como se trabaja con los módulos. Exportar una variable tipo ***string*** con el valor ***'ANTONIO EXPORTADO'*** y mostrarla por consola.

05 Funciones

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Estructura

- Las funciones se pueden usar como en Javascript pero ahora le podemos añadir más cosas como : el tipo de dato devuelto por la función, los tipos de datos de los parámetros de la función, declarar algunos parámetros de la función como opcionales, valores por defecto, vamos a ver estos y alguno más.

- Declaración de la función

function <nombre_función>(<parámetro1>:<tipo_parámetro>,...):<tipo_devuelto>

Ejemplo :

```
function suma(op1:number,op2:number):number{ return op1+op2;}
```

Parámetros opcionales

- **Parámetros opcionales** : podemos declarar parámetros de la función como opcionales con el símbolo **?** tras el nombre del parámetro. Los parámetros opcionales se definen siempre los últimos en la lista de parámetros

Ejemplo :

```
function suma(op1:number,op2?:number):number{ ... }

let suma=(op1:number,op2?:number):number=>{    return p1+p2; }
```

Le estamos indicando a la función que el parámetro **p2** es opcional

Podemos invocar a la función de esta manera

```
suma(12);
suma(12,12);
```

Valores por defecto

- **Valores por defecto en parámetros** : podemos declarar parámetros de la función con valores por defecto. Cuando hay un parámetro con un valor por defecto, por delante de otro parámetro obligatorio, si queremos que coja el valor por defecto hay que invocar a la función con el valor ***undefined*** en el lugar del parámetro con valor por defecto, si le pasamos **null** y estamos realizando una operación numérica lo toma como valor **0**

Ejemplo :

```
function suma(op1:number,op2='12'):number{ ... }
```

Le estamos indicando a la función que el parámetro **op2** tiene un valor por defecto de '12'

Podemos invocar a la función de esta manera

```
suma(12,undefined);
suma(12,12);
```

Ejercicio5

Implementar los ejemplos anteriorres para comprender como se trabaja con las funciones. Probar la declaración de una función , los parámetros opcionales y los valores por defecto de los parámetros.

06 Interfaces

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Asegurando las estructuras de datos

- Nos ayudan definir una estructura de un Objeto , en cuando a los nombre y tipos de sus propiedades.

- Cuando empezamos un proyecto definimos las entidades que interactúan con el sistema, estas entidades tienen unas propiedades y esas propiedades las podemos representar como propiedades de una clase TypeScript con un tipo de dato concreto

- Una interfaz es un conjunto de métodos abstractos y de constantes cuya funcionalidad es la de determinar el funcionamiento de una clase, funciona como un molde o como una plantilla

Asegurando las estructuras de datos

- **Declaración de interface** : usando la palabra clave ***interface*** y añadiendo en el cuerpo del interface las variables con los tipos de datos concretos. La Interfaz se puede usar como tipo de dato , no se puede instanciar.

  Ejemplo :

```
interface datosPersonales{
        edad:number;
        nombre:string;
        apellidos:string;
}
```

  Ahora podríamos usarlo de la siguiente manera :

```
let edad:datosPersonales={
   nombre:'Pepe',
   edad:21
};
edad.edad=12;
```

Asegurando las estructuras de datos

- **Parámetros opcionales en interfaces** : si hay algún parámetro opcional lo marcamos como tal como vimos anteriormente, poniendo detrás del nombre de la variable el símbolo **?**

Ejemplo :

```
interface datosPersonales{
        edad:number;
        nombre:string;
        apellidos:string;
        domicilio?:string;
}
```

Asegurando las estructuras de datos

- **Uso de interfaces en parámetros de funciones**: podemos usar los interfaces en los parámetros de entrada de una función y en el tipo de dato de salida de la función, los usamos como tipos de datos.

Ejemplo : siguiendo el interfaz definido anteriormente

```
function  addDatosPersonales(datos:datosPersonales):void{
   ...
}

let addDatosPersonales=(datos:datosPersonales):void{
   ...
}
```

Ejercicio6

- Crear el interfaz visto en el ejemplo anterior, ***datosPersonales*** y realizar las siguientes tareas :

a) Crear una función llamada ***verDatosPersonales*** que reciba como parámetro una variable con la estructurade ***datosPersonales*** y muestre por consola las propiedades del mismo.

b) Invocar la función con los siguientes parámetros
{edad:100,nombre:'ANTONIO'} ¿Es correcto el resultado?

c) Invocar la función con los siguientes parámetros
'ANTONIO'

d) Invocar la función con los siguientes parámetros
{edad:100,nombre:'ANTONIO',apellidos:'MARTÍNEZ'}

07 Clases

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Usando clases

- Las clases las definimos con la palabra clave ***Class*** seguida del nombre de la clase

Ejemplo:

```
class Cliente{
        edad:number;

        constructor(edad:number){
            this.edad=edad;
        }

        verEdad():void{
            console.log(this.edad);
        }

        devolverEdad():number{
            return this.edad;
        }

}
```

Usando clases

- La estructura es muy parecida a Java, dentro de la clase hay unas propiedades, un constructor y unos métodos.

Ejemplo:

```
Class Cliente{

    edad:number;

    constructor(){
      this.edad=null;
    }

    verEdad():void{
      console.log(this.edad);
    }

    devolverEdad():number{
      return this.edad;
    }
}
```

Propiedades

Constructor

Métodos

Usando clases

- Al igual que en Java podemos establecer los las propiedades o métodos de la clase como públicas (**public*)* , privadas (**private*)* o protegidas (**protected*)*. Estas palabras claves, (**public,private,protected*)* las tenemos que poner delante de las propiedades o métodos que queramos establecer, por defecto si no ponemos nada son declaradas como **public**

- **private** : la propiedad o método que esté declarada como **private** , solo puede ser accedida o invocada desde dentro de la clase, no se puede acceder desde fuera de la clase.

- **public *:*** la propiedad o método que está declarada como **public**, puede ser accedida o invocada desde dentro y fuera de la clase.

- **protected** : igual que **private** , pero también se puede acceder a las propiedades o miembros declarados como **protected** desde las subclases

Usando clases : implements

- Podemos crear una clase e implementarla desde una interface con la palabra clave ***implements.*** Esto nos obligará a que la clase tenga las mismas propiedades que hayamos definido en la estructura del interface.

Ejemplo: crear una clase implementando el interface anterior ***datosPersonales***

```
class Persona implements datosPersonales{

        edad:number;
        nombre:string;
        apellidos:string;
        domicilio?:string;

}
```

Usando clases : extends

- Podemos crear una clase heredada de otra con la palabra clave ***extends***. *Si también usamos* ***implements***, ***extends*** *deberá aparecer siempre a la izquierda e* ***implements*** *a la derecha*

Ejemplo:

```
class Persona{
  edad:number;
 }

class Operario extends Persona{

}

let operario1=new Operario();
operario1.edad=34;
```

Usando clases : extends

- Cuando usamos ***extends*** y usamos el constructor en la subclase (clase hija), tenemos que invocar obligatoriamente el constructor de la superclase (clase Padre) con la palabra clave ***super***

Ejemplo:

```
class Persona{
  edad:number;
  constructor(edad:number){ this.edad=edad; }
}

class Operario extends Persona{
        constructor(edad:number){
          super(edad);
        }
}

let operario1=new Operario();
operario1.edad=34;
```

Ejercicio7

- Crear una clase llamada **Persona** con las siguientes características:

a) Añadir las siguientes propiedades : ***edad*** , ***nombre y apellidos*** como un interface llamado ***datosPersonales***

b) Crear un constructor que reciba un tipo de dato ***datosPersonales*** e inicialice las propiedades de la clase

b) Añadir los siguientes métodos a la clase : ***getEdad, getNombreCompleto***, que devuelvan la propiedad ***edad*** y el nombre completo (***nombre + apellidos)*** respectivamente

c) Crear una clase con las siguientes propiedades ***{edad:100,nombre:'ANTONIO',apellidos:'MARTINEZ GARCIA'})***

Ejercicio8

- Crear una superclase y otra subclase como las del ejemplo anterior (***Persona, Operario)***, implementar la superclase desde la interfaz ***datosPersonales***

a) Añadir los métodos del ***ejercicio7*** a la superclase

b) Probar el funcionamiento del constructor creando una instancia de la subclase ***Operario***

c) Añadir a la superclase varios métodos : ***setEdad, setNombre, setApellidos*** para establecer los valores de las propiedades ***edad, nombre*** y ***apellidos***

d) Invocar los métodos ***setEdad, setNombre, setApellidos*** con cualquier valor y después invocar el método

e) Añadir un nuevo método a la subclase ***Operario*** llamado ***getNombreOperario*** que devuelva 'OPERARIO :' + nombre + apellidos)

Ejercicio9

- A partir del código fuente del ***ejercicio*** realizar las siguientes tareas

a) Crear un método en la subclase ***Operacio*** llamado ***getEdadOperario*** que devuelva la edad del operario e invocarlo

b) Establecer la propiedad ***edad*** de la superclase ***Persona*** como private y volver a llamar al método ***getEdadOperario***

c) Establecer la propiedad ***edad*** de la superclase ***Persona*** como protected y volver a llamar al método ***getEdadOperario***

Usando clases : readonly

- Podemos declarar propiedades de la clase como de sólo lectura con la palabra clave ***readonly***

Ejemplo:

```
class Persona{
  readonly edad:number;
        constructor(edad:number){
          this.edad=edad;
        }
}

class Operario extends Persona{
        constructor(edad:number){ super(edad); }
}

let operario1=new Operario();
operario1.edad=34; // Nos saltaría un error
```

Ejercicio10

- A partir del código fuente del ***ejercicio9*** realizar las siguientes tareas

a) Establecer la propiedad ***edad*** de la superclase ***Persona*** como ***readonly***, comprobar qué pasa con los métodos que usan esta propiedad

Usando clases : static

- Si declaramos una propiedad de una clase como ***static***, esta propiedad solo será visible / accesible por la superclase pero no por las subclases

Ejemplo:

```
class Persona{
  static edad:number;
        constructor(edad:number){
          this.edad=edad;
        }
}

class Operario extends Persona{
        constructor(edad:number){
          super(edad);
        }
}

let operario1=new Operario();
operario1.edad=34; // Nos saltaría un error
```

Ejercicio11

- A partir del código fuente del ***ejercicio10*** realizar las siguientes tareas

a) Establecer la propiedad ***edad*** de la superclase ***Persona*** como ***readonly***, comprobar qué pasa con los métodos que usan esta propiedad

Usando clases : abstractas

- Una clase Abstracta se declara con la palabra clave ***abstract*** delante de la palabra clave ***class***.
- Una clase Abstracta se usa para que otras clases hereden de ella
- No pueden ser instanciadas directamente
- La implementación interna de las propiedades y métodos funciona como los interfaces

Ejemplo:

```
abstract class Persona{
        static edad:number;
        constructor(edad:number){
          this.edad=edad;
        }
 }
class Operario extends Persona{
        constructor(edad:number){
          super(edad);
        }
}
```

Ejercicio12

- A partir del código fuente del ***ejercicio11*** realizar las siguientes tareas

a) Declarar la superclase ***Persona*** como clase abstracta. ¿Qué es lo que sucede?

08 Generics

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Usando clases : reutilizando código

- Cuando hablamos de Generics estamos hablando de componentes reutilizables, se comporta y se usa de manera similar a como ya hemos visto en Java. Cada comodín que le pasamos hace referencia a un tipo de parámetro.

- Se suelen usar comodines como (***T,U,V,W...)*** pero podemos usar otros que describan el parámetro.

Ejemplo :

```
function generica<T>(parametro:T):T{
  return parametro;
}

const generica=<T>(parametro:T):T=>{
   return parametro;
}
```

Podemos invocar la función de varias maneras :

let resultadoCadena=generica<string>('PRUEBA CADENA');

let resultadoNumero=generica<number>(12);

Usando clases : reutilizando código

- Las funciones genéricas nos van a permitir construir una ***interfaz, función*** que acepte cualquier tipo de parámetro que hayamos definido con (***T,U,....***) y nos permitirá construir objetos o funciones con los parámetros que necesitemos en el momento

- Se suele utilizar para reutilizar código

Ejercicio13

- Implementar el ejemplo anterior realizar las siguientes tareas

a) Implementar la función con un tipo de dato ***string*** y probarla

b) Implementar la función con un tipo de dato ***number*** y probarla

c) Implementar la función con un tipo de dato ***Array de números*** y probarla

d) Implementar la función con un tipo de dato ***Array de cadenas*** y probarla

Ejercicio14

- Realizar las siguientes tareas

a) Crear una interfaz genérica llamada ***objetoCliente*** que reciba dos parámetros genéricos y se les asigne a las propiedades ***cliente*** y ***estado*** del interfaz

b) Crear una variable llamada ***cliente1*** creada a través de ***objetoCliente*** asignándole los valores {cliente:'ANTONIO',estado:'ACTIVO'} y mostrarlo por consola.

c) Crear una variable llamada ***cliente2*** creada a través de ***objetoCliente*** asignándole los valores {cliente:12,estado:1} y mostrarlo por consola.

d) Crear un tipo ***enum*** llamado ***CLIENTES*** con dos propiedades (***codigo , estado)***

e) Crear una variable llamada ***cliente3*** creada a través de ***objetoCliente*** asignándole los valores {cliente:CLIENTES.codigo,estado:CLIENTES.estado} y mostrarlo por consola.

Ejercicio15

- Realizar las siguientes tareas

a) Crear una interfaz genérica llamada ***funciones*** que reciba dos parámetros genéricos y se les asigne a las propiedades ***p1*** y ***p2*** de una función que devuelva el primero de ellos

b) Crear una variable llamada ***mayor*** que reciba dos parámetros numéricos y devuelva el máximo valor de los dos.

c) Crear una variable llamada ***f1*** creada a través de ***funciones*** asignándole la función ***mayor*** y realizar la llamada con los valores (***12,13)***

09 Iterables

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Recorriendo objetos

- Para recorrer un objeto iterable podemos usar dos sentencias ***for..in*** o ***for..of***
- ***for..in :*** recorre el objeto y guarda la clave o posición de la propiedad del objeto por cada iteración
- ***for..of :*** recorre el objeto y guarda el valor de la propiedad del objeto por cada iteración
- En ***ES6*** diponemos de dos nuevos objetos iterables : ***Set*** y ***Map***
- **Set (add,has,delete.size):** define un conjunto de valores

```
let aviones=new Set(); aviones.add('BOEING');aviones.add('AVION1');
```

- **Map (set,get,has,delete,size):** define un mapa de clave-valor

```
let animal = new Map();
animal.set('raza','MASTIN');
animal.set('peso','100');
```

Ejercicio16

- Realizar las siguientes tareas

a) Crear un array de string llamado ***animales*** con los valores 'PERRO','GATO','MIEMBRO DE LOS MOJINOS‘ y mostrar los valores usando las sentencias vistas

b) Crear un mapa llamado ***animal*** con los valores ('raza','MASTIN') ('peso','100') y mostrar los valores usando las sentencias vistas

c) Crear un conjunto llamado ***aviones*** con los valores ['BOEING','AVION1'] y mostrar los valores usando las sentencias vistas

10

Funciones Arrow Plantillas

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Lambda vs Arrow

- Podemos definir las funciones con una nueva sintaxis que es similar a la sintaxis lambda usada en Java, son las arrows

- **let <nombre_función> =(<parametro>:<tipo>,...) : <tipo_devuelto> => {**
  <cuerpo_función>
}

Ejemplos*:*

let suma = (op1:number,op2:number):number => { return op1+op2; }

Ejercicio17

- Realizar las siguientes tareas

a) Crear una función suma de dos valores numéricos usando la sintaxis de la función arrow

b) Crear una función producto de dos valores numéricos usando la sintaxis de la función arrow

c) Crear una función concatenar de dos cadenas usando la sintaxis de la función arrow

d) Crear una función que muestre por consola una cadena pasada por parámetro

Plantillas de texto

- Podemos usar plantillas de texto para generar textos con valores de variables

Ejemplo :

```
let valor:number=12;
let plantilla = `Esto es una plantilla con en valor ${valor}`;
```

Ejercicio18

- Realizar las siguientes tareas

a) Crear una plantilla que muestre la variable ***valor:number=12*** dentro del texto

*Esta es una plantilla con valor numérico **12***

b) Crear una plantilla que muestre la variable ***valor:string='HOLA '*** dentro del texto

HOLA , esto es un saludo amistoso

Ejercicio19

- Realizar las siguientes tareas

a) Implementar el ejercicio22 del tema de JavaScript usando TypeScript

b) Añadir una nueva funcionalidad al botón enviar : construir un Objeto JSON con la estructura

```
{
        nombre:"....",
        email:"...",
        provincia:"..."
}
```

* En TypeScript existen una serie de interfaces ***HTML<nombre>*** como ***HTMLElement*** donde podemos guardar los elementos creados dinámicamente o recuperados mediantes ***getElementById***

Ejercicio19

- Realizar las siguientes tareas

c) Cuando se le haga ***click*** al botón enviar ocultar el formulario y mostrar un resumen de los datos recogidos con un botón con el texto ***Confirmar*** y otro con el texto ***Volver***. Cuando se le haga click sobre el botón ***Volver*** se volverá a mostrar el formulario con los datos rellenos , si se hace click sobre el botón ***Confirmar*** no se realizará ninguna acción

Programación Funcional

- La programación funcional se basa en “qué” estamos haciendo en lugar de “cómo” lo estamos haciendo

- [Array].map([función]) : aplica a cada elemento del array la función pasada como parámetro y devuelve un array con el resultado

Ejemplo :

```
let array:Array<string>=['Hola','Hasta Luego','Adios'];

let addAmigo=(elemento)=>elemento+‘ amigo';


console.log(array.map(addAmigo));
```

Programación Funcional

- **[Array].filter([función])** : aplica a cada elemento del array el filtro declarado en la función pasada como parámetro y devuelve un array con el resultado.

Ejemplo :

```
let array:Array<number>=[2,3,4,7,8,10,23];

let filTraPar=(elemento)=>elemento%2==0;

console.log(array.filter(filTraPar));
```

Programación Funcional

- **[Array].reduce([función])** : realiza un cálculo (operación sobre 2 elementos) acumulado sobre cada elemento del array, al primer cálculo sobre el primer elemento se le pasa un valor , el resto de cálculo sobre los otros elementos se realiza con el acumulado del cálculo hasta ese momento. Devuelve el resultado acumulado.

Ejemplo :

```
let array:Array<number>=[1,2,3,4,10];

let suma=(valor1,valor2)=>valor1+valor2;

console.log(array.reduce(suma,0));
```

TypeScript

Referencias

https://www.typescriptlang.org
http://www.ecma-international.org
https://www.typescriptlang.org/docs/handbook/dom-manipulation.html