Jorge García Flores

Desarrollo Web en Entorno Cliente

Tema 19

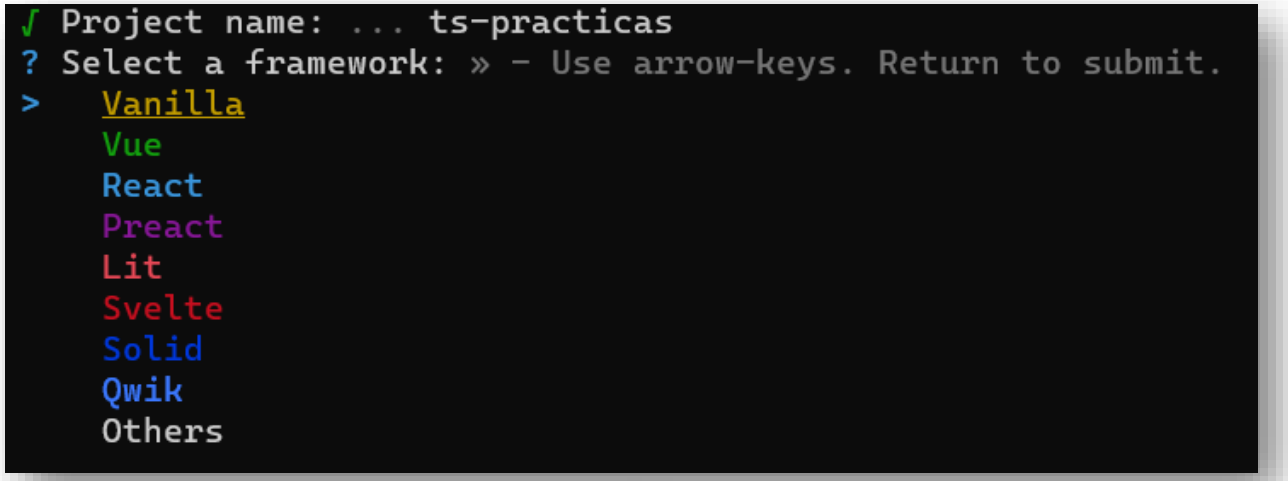
Introducción a TypeScript

Crear un proyecto de TypeScript

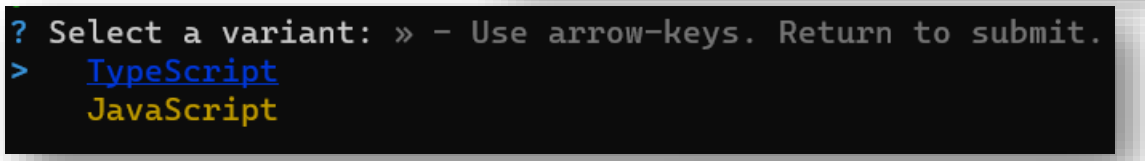
Para crear un proyecto de JS con TypeScript utilizando *npm* y *vite*, primero se crea la aplicación:

```
npm init vite@latest
```

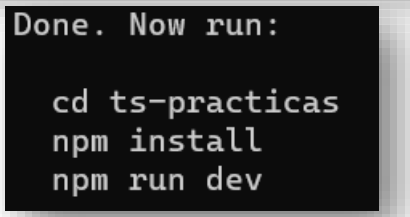
Escribimos el nombre de la misma y seleccionamos a continuación *Vanilla*:



Y después TypeScript:



Por último, seguimos las instrucciones que se indican:



Crear un proyecto de TypeScript

Para crear un proyecto de JS con TypeScript utilizando *yarn* y *vite*, primero se crea la aplicación:

```
yarn create vite
```

Escribimos el nombre de la misma y seleccionamos a continuación *Vanilla*:

```
√ Project name: ... ts-practicas
? Select a framework: » - Use arrow-keys. Return to submit.
>   Vanilla
    Vue
    React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Others
```

Y después TypeScript:

```
? Select a variant: » - Use arrow-keys. Return to submit.
>   TypeScript
    JavaScript
```

Por último, seguimos las instrucciones que se indican:

```
Done. Now run:

  cd vite-project
  yarn
  yarn dev
```

Crear un proyecto de TypeScript

La configuración de TypeScript se encuentra en el archivo *`tsconfig.json`*.

Editando este archivo se puede configurar la versión concreta de *Ecma Script* a la que será transpilado el código.

También se puede configurar el *`bundle`* y el *`linter`*:

tsconfig.json > ...

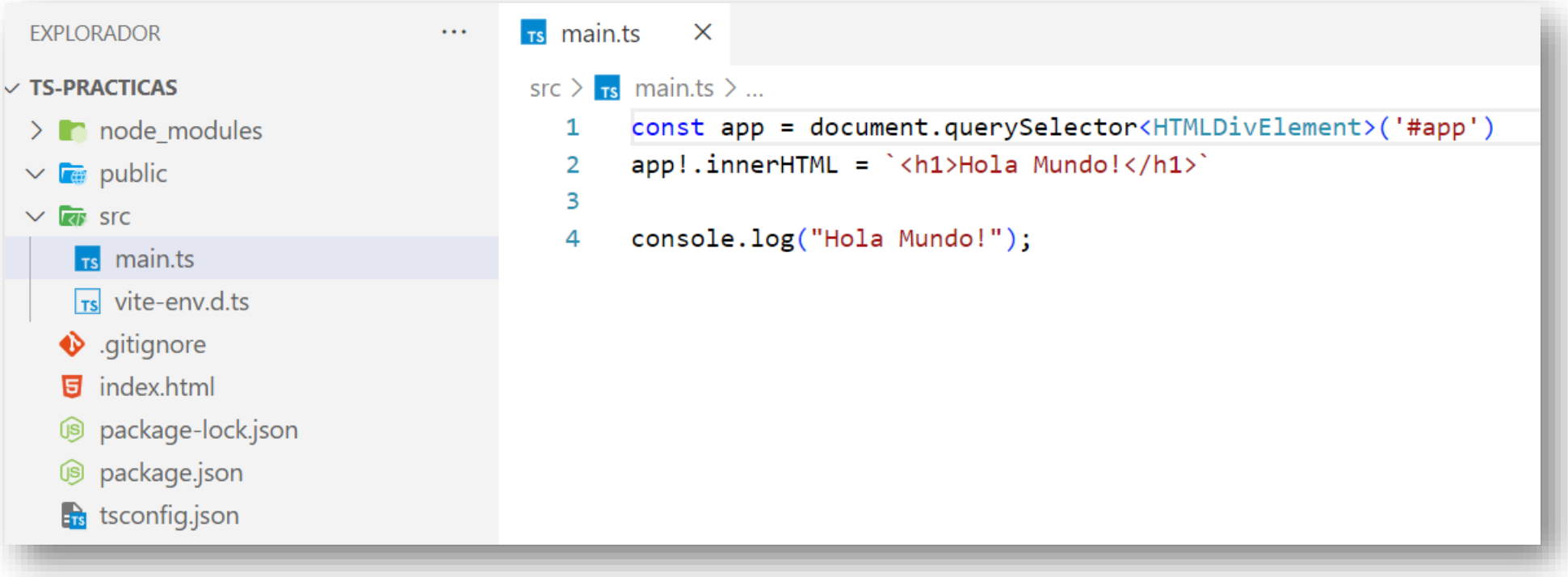
```jsonc
{
  "compilerOptions": {
    "target": "ES2020",
    "useDefineForClassFields": true,
    "module": "ESNext",
    "lib": ["ES2020", "DOM", "DOM.Iterable"],
    "skipLibCheck": true,

    /* Bundler mode */
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,

    /* Linting */
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noFallthroughCasesInSwitch": true
  },
  "include": ["src"]
}
```

Crear un proyecto de TypeScript

Por último, se pueden borrar los archivos de ejemplo que vienen con *vite* para dejar la aplicación lista para empezar a desarrollarla:



Tipos de datos

Si no se especifica un tipo para una variable TypeScript va a inferir su tipo.

En el siguiente ejemplo, se crea una variable y se le asigna el valor de un *string* por lo que TypeScript infiere que es de tipo *string*, más adelante se quiere asignar a dicha variable un número y nos da el error asociado a que se está asignando un tipo *number* a una variable de tipo *string*:

```
let nombre ='Jorge';

nombre= 123;     El tipo 'number' no se puede asignar al tipo 'string'.
```

Este error no aparecía en JS ya que era de tipado débil y dinámico.

Para las constantes, les crea un tipo de datos que se llama igual que el nombre de la constante, al fin y al cabo, una constante nunca puede cambiar de valor y de tipo una vez declarada:

```
const apellido: "Garcia"
const apellido = "Garcia" ;
```

Tipos de datos

Para declarar una variable de un tipo de datos, se declara, a partir del nombre de la misma, con `:` y el tipo de datos que se desea. Por ejemplo:

```
let nombre : string = 'Jorge';
```

Los tipos de datos que se pueden declarar son:

- *string*
- *number*
- *boolean*
- *void*
- *any*
- *unknown*
- *enum*
- *never*
- *undefined*
- *object* un tipo de datos no primitive
- *null*
- Tuplas como *[string, number]*
- *[]* o un array de un tipo como *Array<elemType>* o *number[]*
- o incluso uno personalizado, como en el ejemplo pasado de las constantes

Tipos de datos

Para especificar que una variable puede contener más de un tipo de datos se utiliza el símbolo de la tubería | entre los diferentes tipos de datos que puede contener.

En el siguiente ejemplo, se crea una variable llamada precio que puede contener el tipo de datos *string* y *number*:

```
let precio : string | number = '10 €';
precio = 12;
```

Otro ejemplo:

```
let precios : number | number[] | string;
precios = 0;
precios = [12, 20, 40];
precios = "12 20 40";
```

Tipos de datos

Además, se puede definir un tipo de datos personalizado:

```
let deposito : "Lleno" | "Medio" | "Vacío";
deposito = "Lleno";
```

Para "tipar" objetos se utilizan las interfaces. Una interfaz no existe en el lenguaje JS, solo existe en TypeScript, por lo que no será transpilada.

La interfaz se define igual que un objeto, pero se antepone la palabra *interface*. Dentro de la interfaz se especifican las propiedades junto con sus tipos de datos:

Si alguna propiedad es opcional se utiliza el símbolo de interrogación *?* junto con el nombre de la propiedad:

```
interface Persona {
    nombre: string;
    apellidos: string;
    edad: number;
    hijos?: Persona[];
}

const Jorge: Persona = {
    nombre: "Jorge",
    apellidos: "García Flores",
    edad: 0
}
```

Tipos de datos

A veces, se utiliza el tipo de datos *undefined* como un posible tipo de la propiedad:

Pero se obtiene un error si no se aparece en la declaración del objeto:

```
interface Persona {
    nombre: string;
    apellidos: string;
    edad: number;
    hijos: Persona[] | undefined;
}

const Jorge: Persona = {
    nombre: "Jorge",
    apellidos: "García Flores",
    edad: 0,
    hijos: undefined
}
```

```
interface Persona {
    nombre: string;
    apellidos: string;
    edad: number;
    hijos: Persona[] | undefined;
}

const Jorge: Persona = {    La propiedad "hijos" falta en el tipo
    nombre: "Jorge",
    apellidos: "García Flores",
    edad: 0
}
```

Funciones

Cuando se define una función TypeScript trata de inferir los tipos de retorno y de los argumentos de la misma:

Si la función no devuelve nada el tipo de dato devuelto es *void*.

```
function obtenerNombre(p: Persona): string
function obtenerNombre(p:Persona){
    const {nombre}=p;
    return nombre;
}
```

Para especificar un tipo de datos de retorno en una función se colocan los dos puntos **:** a continuación del paréntesis de cierre de los argumentos y seguido los tipos de datos de retorno:

```
function obtenerNombre (p:Persona) : string {
    const {nombre}=p;
    return nombre;
}
```

Funciones

El tipo de datos *any* se utiliza para especificar que el tipo que admite el argumento puede ser cualquiera:

```
function sumar (a: any , b: any) : any {
    return a + b;
}

sumar (2,4);
sumar ("Hola", " Mundo");
```

Aunque se debe evitar utilizar *any* ya que hace que se pierda parte de la utilidad de TypeScript:

```
function sumar (a: any , b: any) : any {
    return a + b;
}

const resultado: string = sumar ([1, "a"],[2, "c"]);
console.log(resultado);
```

Funciones

También se puede especificar que los argumentos sean opcionales utilizando el símbolo *?*:

```
function sumar (a: number, b?: number) : number {
    return a + b;     "b" es posiblemente "undefined".
}
```

Pero TypeScript indicará que si no se pasa ningún argumento será de tipo *undefined*.

Además, se pueden utilizar valores por defecto, para ello se coloca un = seguido del valor que se quiere asignar al argumento en caso de que no se proporcione justo después del tipo de datos:

```
function sumar (a: number=0, b: number=0): number
{
  return a + b;
}
```

Funciones

Se puede especificar como argumento de una función una interfaz. Esto asegura que se utilizaran las propiedades definidas dentro de la interfaz:

```
interface Persona {
  nombre?: string;
  apellidos?: string;
  edad?: number;
  hijos?: Persona[];
}
```

```
function establecerEdad(persona: any, años: number=0): void
{
  persona.años ??= 0;
  persona.años += años;
}
```


```
function establecerEdad(persona : Persona, años: number=0): void
{
  persona.edad ??= 0;
  persona.edad += años;
}
```

Funciones

Se pueden utilizar funciones dentro de clases u objetos literales, en este caso hay que recordad que no se llaman funciones sino métodos. Para crear un método dentro de un objeto literal, se pone el nombre del método, a continuación *: function* (o una función flecha) y después la definición de la función como se ha visto en las páginas anteriores:

```
const Jorge = {
  saludar: function (nombre: string=""): void {console.log(`Hola ${nombre}`);}
}
```

También se puede hacer omitiendo *: function*:

```
const Jorge  = {
  saludar (nombre: string =""): void { console.log(`Hola ${nombre}`);}
}
```

Está última manera es la que se utiliza para crear un método dentro de una clase:

```
class Persona {
  saludar (nombre: string=""): void {console.log(`Hola ${nombre}`);}
}
```

Métodos en interfaces

Para utilizar métodos dentro de interfaces se suelen utilizar funciones flecha. Para ello se pone el nombre del método seguido de *: ()=>* y el tipo de datos que devuelve dicho método.

En el ejemplo de la derecha se ha añadido el método saludar a la interfaz *Persona*, este método no devuelve *void* (nada).

```
interface Persona {
  nombre: string;
  apellidos: string;
  edad: number;
  hijos?: Persona[];
  saludar: () => void;
}
```

Otra forma de especificar un método dentro de la interfaz es colocando el nombre del método (se puede omitir) seguido de los argumentos que recibe dicho método y los *:* y el valor devuelto:

```
interface Persona {
  nombre?: string;
  apellidos?: string;
  edad?: number;
  hijos?: Persona[];
  saludar(): void;
  asignarNombre (nombre: string): void
}
```

Clases

TypeScript añade una sintaxis más rica a la definición de las clases que la que tiene JS.

Se puede utilizar los modificadores de acceso *public* y *private* para definir propiedades y métodos públicos y privados en la clase.

Además, se puede utilizar la palabra reservada *readonly* para crear una "constante" dentro de la clase. Es una variable que una vez inicializada no pueda cambiar su valor.

Aunque, no obstante, al transpilarse a código JS todas estas funcionalidades no existen, por lo que carecerán de estas características.

```
class Persona {
  private nombre: string;
  private readonly apellidos: string;
  public hijos?: Persona[];

  constructor (nombre: string, apellidos: string)  {
    this.nombre = nombre;
    this.apellidos=apellidos;
  }
  get Nombre (): string {return this.nombre;}
  set Nombre(nombre: string) { this.nombre= nombre;}
  get Apellidos (): string {return this.apellidos;}
}

const Jorge = new Persona ("Jorge", "García Flores");
console.log(Jorge.Apellidos);
```

Clases

También se puede colocar en el constructor la definición de las propiedades con sus modificadores de acceso, reduciendo considerablemente el código.

```
class Persona {
  //private nombre: string;
  //private readonly apellidos: string;

  constructor (private nombre: string, private readonly apellidos: string)  {
    this.nombre = nombre;
    this.apellidos=apellidos;
  }
  get Nombre (): string {return this.nombre;}
  set Nombre(nombre: string) { this.nombre= nombre;}
  get Apellidos (): string {return this.apellidos;}
}

const Jorge = new Persona ("Jorge", "García Flores");
console.log(Jorge.Apellidos);
```

Herencia

Mediante el uso de las interfaces y las clases e incluso otros tipos de datos como los *enum* se pueden crear clases más complejas haciendo uso de la herencia.

El siguiente ejemplo, realizado únicamente con fines didácticos, ilustra estas estructuras más elaboradas:

No obstante, como buena práctica de programación no se recomienda utilizar la herencia más allá de tres niveles, ya que en ese caso la legibilidad y la comprensión del código resulta muy compleja. En este caso se recomienda utilizar la composición.

```
enum Sexo {
  macho = "macho",
  hembra = "hembra"
}

interface Animal {
  sexo: Sexo
}

class SerVivo implements Animal{
  sexo: Sexo = Sexo.macho;
  respirar (): void {console.log("respirando");}
}

class Persona extends SerVivo {
  nombre?: string;
  apellidos?: string;
  edad?: number;
  hijos?: Persona[];
}

const Jorge = new Persona();
```

Genericidad

Es una característica de los lenguajes de programación que permite realizar una funcionalidad no solo para un tipo de datos, sino para todos los tipos que se quiera.

La genericidad consiste en una serie de técnicas que permiten aplicar cierta función o algoritmo a un amplio rango de tipos de datos.

Se realiza básicamente de dos maneras:

- Haciendo abstracción del tipo de datos que contienen o al que son aplicados.
- Parametrizando el tipo o tipos de datos que intervienen.

Lenguajes como C# y Java utilizan la genericidad para crear componentes reutilizables, permitiendo crear un componente que pueda funcionar con una variedad de tipos en lugar de uno solo.

Genericidad

En TypeScript se puede crear genericidad utilizando el tipo de datos *any*, sin embargo, esto anula la ventaja de utilizar el tipado de TypeScript.

En el siguiente ejemplo la función identidad solo puede utilizarse con argumentos de tipo *number*.

```
function identidad(arg: number): number {
  return arg;
}
```

Al utilizar el tipado *any* conseguimos que la función sea genérica para todos los tipos de datos.

```
function identidad(arg: any): any {
  return arg;
}
```

Con *any* se pierde la información sobre cuál era el tipo de datos que devuelve la función. TypeScript dispone de la palabra *Type* un tipo de datos especial que funciona con tipos en lugar de valores.

```
function identidad<Type>(arg: Type): Type {
  return arg;
}
```

Genericidad

En TypeScript se puede crear genericidad utilizando el tipo de datos *any*, sin embargo, esto anula la ventaja de utilizar el tipado de TypeScript.

En el siguiente ejemplo la función identidad solo puede utilizarse con argumentos de tipo *number*.

```
function identidad(arg: number): number {
  return arg;
}
```

Al utilizar el tipado *any* conseguimos que la función sea genérica para todos los tipos de datos.

```
function identidad(arg: any): any {
  return arg;
}
```

Con *any* se pierde la información sobre cuál era el tipo de datos que devuelve la función. TypeScript dispone de la palabra *Type* un tipo de datos especial que funciona con tipos en lugar de valores.

```
function identidad<Type>(arg: Type): Type {
  return arg;
}
```

Genericidad

Se puede especificar o “tipar” el argumento que recibe una función genérica.

Para ello se especifica el tipo de datos entre los símbolos <> justo antes de los paréntesis.

Por ejemplo, el método *querySelector* de *document* devuelve un *Element*, mediante la genericidad se especifica el tipo que devolverá dicha función:

```
const app = document.querySelector<HTMLDivElement>('#app');
```

Suele ser habitual utilizar para el primer tipo de datos usar la letra T, para el segundo U, para el tercero V…

```
function identidad<T, U, V>(arg1: T, arg2: U, arg3: V) {
  return {arg1, arg2, arg3};
}
```

Genericidad

La genericidad aporta muchas ventajas y se utiliza para muchas cosas:

Se puede utilizar con la palabra reservada *Type* para crear un nuevo tipo de datos, un envoltorio (wrapped).

```
type Wrapped<T> = { value: T };
const wrappedValue: Wrapped<number> = { value: 10 };
```

O como en el siguiente ejemplo, donde se utiliza la genericidad para que la función llamada *primerElemento* devuelva el primer elemento de un array que se le pasa como argumento; gracias a ella la función devuelve el tipo de datos correcto en función de los tipos de datos del array.

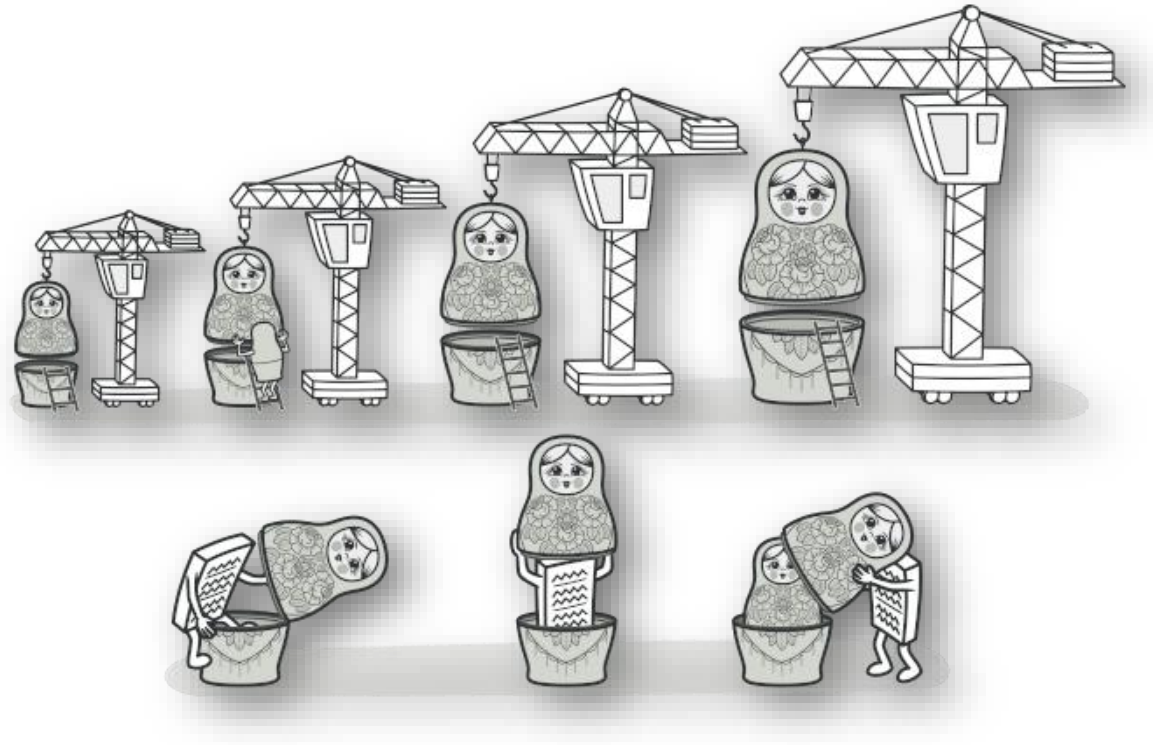
```
function primerElemento<T>(arr: T[]): T {
  return arr[0];
}

const numberArray: number[] = [1, 2, 3, 4, 5];
const stringArray: string[] = ['manzana', 'banana', 'pera'];

const primerNumero = primerElemento<number>(numberArray);
const primeraCadena = primerElemento<string>(stringArray);
```

Decoradores

Los decoradores se utilizan con frecuencia en los diferentes *frameworks* de JS, como en Angular.



Un decorador (*decorator*) es un patrón de diseño estructural, que permite al programador añadir nuevas funcionalidades a un objeto existente sin alterar su estructura, mediante la adición de nuevas clases que envuelven a la anterior dándole funcionamiento extra.

Wrapper (envoltorio) es el sobrenombre alternativo del patrón decorador, que expresa claramente su idea principal. Un *wrapper* es un objeto que puede vincularse con un objeto objetivo. El *wrapper* contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe.

No obstante, el *wrapper* puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.

Decoradores

Los decoradores en TypeScript no son más que funciones que se adjuntan, que envuelven, a diferentes clases, métodos, propiedades...

Estas funciones proporcionan nuevas características (la “funcionalidad” que realiza la función) sin alterar la estructura o contenido del mismo.

Los decoradores, a pesar de que están siendo utilizados desde hace más de 13 años y son muy estables, TypeScript los considera una característica experimental, por lo que hay que habilitarla en el fichero de configuración de TypeScript del proyecto, tsconfig.json.

En este fichero hay que añadir dentro de la sección *"compilerOptions"* la siguiente propiedad:

```
"experimentalDecorators": true
```

```
{
  "compilerOptions": {
    "target": "ES2020",
    "useDefineForClassFields": true,
    "module": "ESNext",
    "lib": ["ES2020", "DOM", "DOM.Iterable"],
    "skipLibCheck": true,
    "experimentalDecorators": true,
```

Decoradores

Para usar un decorador se pone @ y el nombre del decorador justo delante del elemento a "decorar".

A continuación, se ilustra un ejemplo sencillo del funcionamiento de un decorador.

En este caso llamado *spanishDecorator* añade la propiedad nacionalidad con valor española a la clase Persona cada vez que se crea un nuevo objeto dicha clase sin haber alterado su estructura interna.

```
function españolaDecorator(target: typeof Persona): typeof Persona {
    return class extends target {nacionalidad: string = "española";}
}

@españolaDecorator
class Persona {
  constructor (private nombre: string, private readonly apellidos: string)  {
    this.nombre = nombre;
    this.apellidos=apellidos;
  }
  get Nombre (): string {return this.nombre;}
  set Nombre(nombre: string) { this.nombre= nombre;}
  get Apellidos (): string {return this.apellidos;}
}

const jorge = new Persona("Jorge", "Garcia Flores");
console.log(jorge);
```

De igual manera, el decorador, por ejemplo, podría haber añadido varios métodos a la clase.

Non-null assertion

El operador *!*, conocido como "*non-null assertion*", se utiliza para indicar al editor y al transpilador que el valor de la expresión nunca será *null* o *undefined*.

No obstante, esto es una manera de "engañar" al editor y al transpilador para que no muestre ningún error; por eso, siempre es una buena práctica verificar si el valor no es un *nullish* antes de usar *!* con el fin de evitar errores inesperados.

En el ejemplo de la derecha se utiliza el operador *!* para indicar que *app* existirá.

Sin esto, el editor avisaría que *app* puede ser *null* o *undefined*, ya que podría suceder que al hacer el *querySelector* de un elemento con id *app* no hubiera ninguno en la página HTML.

```
const app = document.querySelector<HTMLDivElement>('#app')

app!.innerHTML = `<h1>Hola Mundo!</h1>`

console.log("Hola Mundo!");
```

Bibliografía y recursos online

- https://www.typescriptlang.org/docs/handbook/intro.html
- https://es.vitejs.dev/guide
- https://www.typescriptlang.org/docs/handbook/2/generics.html
- https://microsoft.github.io/PowerBI-JavaScript/interfaces/_node_modules_typedoc_node_modules_typescript_lib_lib_dom_d_.htmldivelement.html
- https://www.w3schools.com/typescript/index.php

Bibliografía y recursos online

- https://www.cosmiclearn.com/lang-es/typescript-index.php
- https://refactoring.guru/es/design-patterns/decorator
- https://www.typescriptlang.org/docs/handbook/decorators.html
- https://refine.dev/blog/typescript-decorators/#method-decorators-in-typescript
- https://blog.logrocket.com/practical-guide-typescript-decorators/