**TYPESCRIPT**

# INSTALACIÓN.

Configurará su equipo de acuerdo con su plataforma y necesidades específicas. Los usuarios de Windows y Visual Studio simplemente deben descargar el Plugin de Visual Studio. Si utiliza Windows y no tiene Visual Studio, puede probar Visual Studio Express for Web.

Actualmente, la experiencia de TypeScript en Visual estudio es superior a la de otros editores de código.

Si utiliza una plataforma diferente (o no desea usar Visual Studio), todo lo que se necesita para usar TypeScript es un editor de texto, un navegador y el paquete npm TypeScript.

Siga las siguientes instrucciones:

1.Instalar el Gestor de Paquetes de Node (npm)

$ curl http://npmjs.org/install.sh | sh

$ npm –version 1.1.70

2.Instalar el paquete npm TypeScript

$ npm install -g typescript

Con esto se instala la versión más reciente de typescript

$ npm view typescript version

npm http GET https://registry.npmjs.org/typescript

npm http 304 <https://registry.npmjs.org/typescript> 0.8.1-1

Para compilar nuestros ficheros .ts en la terminal:

tsc nombreFichero.ts

La extensión del archivo se puede omitir

La herramienta TSC incluye además "watchers" que permiten vigilar cambios en los archivos TS,

tsc -watch miArchivo.ts

Si queremos que no nos compile cuando encuentre errores:

tsc --noEmitOnError miArchivo.ts

Compilando el código a JS en el instante sin que tengamos que intervenir.

Ya tendremos el archivo .js listo. Ahora es cuestión de linkar el .js a nuestra página WEB.

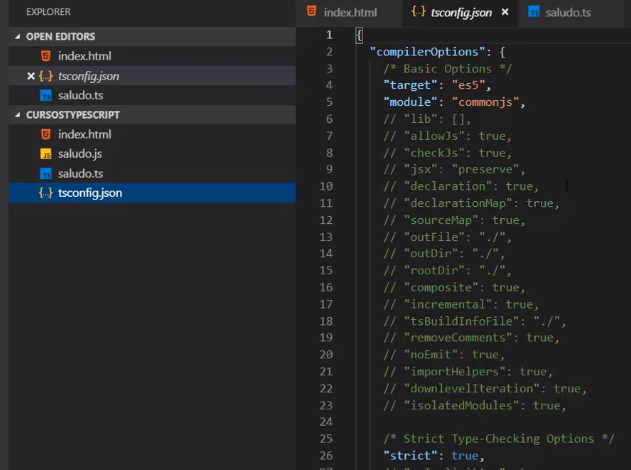
**Importante**: Hay que linkar el .js resultante de la compilación, no el .ts

Además, es habitual que el compilador, se use en conjunto con otras herramientas como Gulp para mejorar aún más el flujo de desarrollo.

Existe asimismo un archivo de configuración llamado "tsconfig.json" en el que podemos indicar todas las opciones que queramos para el compilador (por ejemplo el estándar EcmaScript al que deseamos compilar el código, el tipo de reporte de errores que se desea, las rutas donde colocar los archivos compilados, etc).

Si hacemos la compilación de forma manual tendremos que crear el archivo json. Para ello escribiremos en la consola:

tsc -init



Un sitio interesante para probar TypeScript sin necesidad de instalar nada, o para revisar rápidamente cómo un código TS compilaría a Javascript, es el [TypeScript Playground](https://www.typescriptlang.org/play?#code/FAAjOYEsDsBcCmAnAZgQwMYJASQLID2cBSIA3qOAM5xpJwAUAlANyVg0EAOzbAvpAwAbNFSohCxJAGECAWwBGAVxpQCMEFDlchCOQnjj8REuXYg5dDGgBcIGklgBzNuAsEAJgiEE7D567gaACPviAwSorIgWBwUFwEGPLKqgq6fnCOMC6QbkkwDkoYUgxyGVlOADTuHuXO1bbhkQrI1Rh12Uxmbm5wABZQVAB0lkjWIAC8FjE9-YMjnt4EkzUzvQPDIctTaGvgc8NxCUmKKnFp2FMYM3xs5jR0jF0UPeD5VAS6Qz5ODABEdCQaBg4287ikf2qBxGVjQ1QA5CB4VCNgsvD4EUiUfMtqxzLdctRiDxnuY8uoPl8fv8uHQ0CAwXITEhISBoaNrJjkWzUUz0QQudjNqE8W4CQJgMJROJJCQAKK6YqOJKabS6fSGXCy0gvcAcxr+bIzPlLDo5NxbOwRKJIGbW5AEBRoRCOURW5rRQlgd6ZIolMr2TL1Gpmhrum3VBThlpIUmvWK82ErOR7BPzE0+ZOeVM8nGhFa7Mn7VH2pCO53IKCiFYKG53NwPejMbrx96fBDfAi-AFIIEg+mM5kMxVBpKs9mwwW54YZgVIrHToa4utewPcZu615tqldml04fg0zeBBKqBjoUwsZw+fc9mLDE3i-L-H8SC6OAWLRDqYwBAAdwkZlZFOVR1AYeF9DGBAvCobl4SoABmAAGbkAEYAE50IAVgRDwoAQKhvHhPE5C-KQhkbJ42FIplyM4Ek2HfNlIm-cJ-0AqQFRPUcCHAxAqBEOCMxABDuQAJiQjDqjErC8TgFi6NoJs8VASA1PUoA).

Fuentes de información oficiales:

www.typescriptlang.org

typescript.codeplex.com (lugar de descarga)

go.microsoft.com (especificación en PDF)

www.ecmascript.org

www.ecma-international.org

# TIPOS BÁSICOS

Al ser *TypeScript* un lenguaje fuertemente tipado, a diferencia de JavaScript debemos definir el tipo de dato en la declaración de la variable, evitando así errores posteriores con los tipos de datos y detectándolos en el propio desarrollo.

En *JavaScript*, al contrario, la comprobación de los tipos se realiza durante la ejecución del programa.

En *TypeScript*, al igual que en *JavaScript*, podemos declarar tres tipos de variables:

***var***, ***let*** y ***const***. Recordemos que estos tipos estaban íntimamente relacionados con los ámbitos. Con la variable ***var*** estaríamos trabajando con una variable global, con la variable ***let*** estaríamos con una variable local en el ámbito de un bloque y con la variable ***const*** podríamos asimilarlo a la idea de constante.

**Sintaxis**:

var nombreVariable: tipoDato = valorInicial;

Como hemos visto la palabra reservada **var** podría ser también **let** o **const** dependiendo del ámbito.

Ejemplo**:**

var cadenaGlobal: string = 'Hola mundo!';

let cadenaLocal: string = 'Hola mundo!';

const cero: number = 0;

## Null, undefined y void

Al igual que en JavaScript, el tipo de datos ***null*** tiene un único valor válido ***null***.

Ejemplo**:**

let a: null = null;

El tipo ***undefined*** aparece cuando trabajamos con una variable de la que desconocemos el tipo y no ha sido definida. Si definimos como tipo de variable ***undefined***, el único valor posible es ***undefined***.

Ejemplo**:**

let b: undefined = undefined;

Por defecto tanto ***null*** como ***undefined*** son subtipos del resto de tipos de datos, es decir, que podemos asignar tanto ***null*** como ***undefined*** a cualquier otro tipo de dato.

Ejemplo**:**

let c: number = null;

El tipo de datos ***void*** se utiliza sobre todo en funciones de las que podemos obviar su valor de retorno. Cuando lo usamos en variables, los únicos dos valores posibles para la misma son ***null*** o ***undefined***.

Ejemplo**:**

let d: void = undefined;

let e: void = 1; // Error

## Boolean

Como sabemos, los datos booleanos pueden tomar dos valores ***true*** o ***false***:

Ejemplo**:**

let activo: boolean = false;

En typescript ***true*** y ***false*** van sin comillas.

## Number

Como en JS, todos los números en ***TypeScript*** se almacenan internamente en formato coma flotante. También permite el uso de formatos decimales, hexadecimales, binarios y octales. Además, como en JS, existen valores particulares: ***+Infinity, -Infinity***y***NaN***.

Ejemplo**:**

let a: number = 3;

let b: number = 0b1001; // binario

let c: number = 0o246; // octal

let d: number = 0xf0d; // hexadecimal

## String

La sintaxis a la hora de trabajar con las cadenas de texto es la misma que en JS, pudiendo utilizar para delimitarlas tanto la comilla simple, ‘, como la doble, “, o la comilla invertida, `.

Los ***string*** delimitados por estas comillas invertidas se denominan ***template string***.

Estos nos permiten separar la cadena de texto en varias líneas, y en ***TypeScript*** sobretodo, nos permiten embeber expresiones dentro de la cadena de texto.

Ejemplo**:**

let cadena:string ="Hola mundo!";

let cadena2:string ='Hola mundo!';

let nombre: string = `Juan`;

let saludo: string = `Hola, mi nombre es ${ nombre }`//en TS

var saludo = "Hola, mi nombre es " + nombre; // en JS

## Array

En ***Typescript*** tenemos dos opciones para representar los ***arrays***:

nombreArray: tipoDato[]=[valor1, valor2…valorN];

nombreArray: Array<tipoDato>=[valor1, valor2…valorN];

Ejemplo**:**

let arrayString: string[] = ['a', '2', 'b'];

let arrayNumero: Array<number> = [1, 2, 3];

Podemos tener ***arrays*** con tipos de datos mezclados.

Ejemplo**:**

let arrayMezclado: any[] = [2, 'a', 3]; // en TS

var arrayMezclado = [2, 'a', 3]; //en JS

## Tupla

Es un tipo particular de ***array***, nos permite fijar el tipo de algunos (no tienen por qué ser todos) elementos del ***array*** de antemano.

**Ejemplo:**

let tabla: [number, string];

tabla=[5, 'Hola Mundo']; //ok

tabla=[5,6]; //error

## Enum

Los tipos de dato ***enum*** nos permiten crear colecciones. Este tipo de dato se usa en varios lenguajes de programación, como *C* o *Java*. No estaba disponible en *JavaScript*, pero sí que está disponible en *TypeScript*.

Ejemplo **en TS:**

**3. Tipos Básicos**

enum granja {vaca, caballo, cerdo, gallina, pato}

let a: granja = granja.cerdo;

console.log(granja[3]); // gallina

console.log(granja.caballo); //1

El mismo ejemplo anterior compilado en JS:

var granja;

(function (granja) {

granja[granja["vaca"] = 0] = "vaca";

granja[granja["caballo"] = 1] = "caballo";

granja[granja["cerdo"] = 2] = "cerdo";

granja[granja["gallina"] = 3] = "gallina";

granja[granja["pato"] = 4] = "pato";

})(granja || (granja = {}));

var a = granja.cerdo;

console.log(granja[3]); // gallina

console.log(granja.caballo); //1

Por defecto, la enumeración comienza en 0 pero se puede modificar de forma manual, tanto la posición de partida como todas las posiciones de la colección.

Ejemplo:

enum granja {vaca=3, caballo=1, cerdo=2, gallina=5, pato}

let a: granja = granja.cerdo;

console.log(granja[3]); // vaca

console.log(granja[4]); // undefined

console.log(granja[6]); // pato

console.log(granja.caballo); //1

## Any y Never

Tal y como indica su nombre ***Any*** nos permite definir variables de cualquier tipo.

Ejemplo:

let arrayMezclado: any[] = [2, 'a', 3];

Si no especificamos un tipo al declarar una variable se tomará automáticamente como de tipo ***any***:

let miVariable;

El compilador lo toma como si fuese:

let miVariable:any;

Para el caso opuesto, cuando queremos representar valores que nunca ocurren, por ejemplo, el tipo de una expresión que siempre lanza una excepción o el tipo de retorno de una función que nunca retorna, se utilizaría ***never***.

Ejemplo:

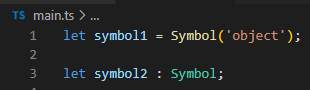
function error(mensajeError: string): never {

throw new Error(mensajeError);

}

## Symbol

Nuevo en ECMAScript 6 por lo no es compatible con versiones inferiores a ES6. Nos permite crear objetos únicos.

[](http://javadesde0.com/wp-content/uploads/Screenshot_29-1.png)

# TIPOS AVANZADOS

TypeScript dispone de la funcionalidad de crear nuestros propios tipos personalizados mediante el uso de la palabra reservada **type**.

Ejemplo:

    //creamos un nuevo tipo de dato: color

    type color = "blue" | "green" | "red" | "yellow";

    function colorea(c: color) {

        let elegido: string = 'El color es ' + c;

        console.log(elegido);

    }

    colorea("red");

## Aserción de tipo

TypeScript nos permite forzar que una variable se comporte como un tipo de dato concreto, independientemente del tipo de dato que sea. Esto se denomina en inglés “Type Assertion”.

Existen dos operadores distintos que nos dan el mismo resultado:

**4. Tipos Avanzados**

• **Operador <>.** Indicando entre **<>** el tipo de dato al que vamos a forzar seguido del nombre de la variable que vamos a forzar.

**Ejemplo:**

    let indeterminado:any = "Hola";

    let longitud:number = (<string>indeterminado).length;

    //*length* no existe para *any* pero sí para *string*

    console.log(longitud); //4

• **Operador as**. En este caso la sintaxis es: nombre de la variable a forzar seguido de la palabra reservada **as** y del tipo de dato al que vamos a forzar.

Ejemplo**:**

    let indeterminado:any = "Hola Mundo!";

    let longitud:number = (indeterminado as string).length;

    console.log(longitud); //11

## Unión de tipos

Para una misma variable podemos indicar que puede almacenar más de un tipo de datos, usando Union Types (Unión de Tipos). Esto se hace a la hora de indicar el tipo, poniendo los dos tipos posibles separados por el operador **|**.

Veámoslo con un ejemplo.

    let edad: number | string;

    edad=32;

    console.log(edad); //32

    edad='treintaydos';

    console.log(edad); // treintaydos

## Intersección de tipos

La intersección de tipos combina múltiples tipos en uno solo. La sintaxis es poniendo los dos tipos separados por el operador **&**. Esto nos resultará especialmente útil en la POO con objetos que implementen más de una interfaz.

Ejemplo**:**

    interface Persona {

        nombre: string;

    }

    interface Telefono {

        numero: number;

    }

let contacto: Persona & Telefono = { nombre: "Juan",

numero: 655655655 };

console.log(contacto); // Object { nombre: "Juan", numero: 655655655 }

## Generics

Una de las novedades que introduce TypeScript respecto a JavaScript son los genéricos (generics). Los genéricos son una utilidad que nos permite reutilizar código, una plantilla que nos permite utilizar un mismo código para varios tipos de variables distintos.

TypeScript nos permitiría hacer esto usando el tipo **any** pero si lo hiciéramos así perderíamos las ventajas del control de tipos que nos ofrece TypeScript. La solución pasa por el uso de genéricos.

**Sintaxis**

Para indicar el uso de un tipo genérico se utiliza el nombre (alias) que le demos al tipo genérico que vayamos a utilizar escrito entre los símbolos **< >**. Este alias de tipo nos permite realizar un control de tipos sobre este tipo genérico que hemos añadido.

Como tipo genérico podemos encontrarnos cualquier tipo de dato, incluso una clase o un array. Incluso podríamos tener varios genéricos en una misma función, deberíamos declarar ambos alias dentro de los mismos **< >** y separarlos entre ellos por comas.

Estas plantillas de genéricos las podríamos utilizar en funciones, tipos, clases…

Veámoslo con algún ejemplo:

Ejemplo función genérica:

//Sin genéricos

function mostrarNumero(x: number): number {

    console.log(x);

    return x;

}

mostrarNumero(12);

function mostrarString(x: string): string {

    console.log(x);

    return x;

}

mostrarString('doce'); //doce

//Usando any

function mostrarAny(x: any): any {

    console.log(x);

    return x;

}

mostrarAny(12); //12

//Usando genéricos

function mostrar<tipoG1>(x: tipoG1): tipoG1 {

    console.log(x);

    return x;

}

mostrar(12); //12

mostrar('doce');//12

//Ejemplo clase genérica:

class Generica<TipoC> {

    suma: (x: TipoC, y: TipoC) => TipoC;

}

let objetoGenerica1 = new Generica<number>();

objetoGenerica1.suma = function (x, y) {

    return x + y;

}

let objetoGenerica2 = new Generica<string>();

objetoGenerica2.suma = function (x, y) {

    return x + y;

}

let a = objetoGenerica1.suma(30, 14);

console.log(a); //44

let b = objetoGenerica2.suma("Hola ", "Mundo!");

console.log(b); //Hola Mundo!

//Ejemplo dos tipos genéricos:

function mostrarTipo<TipoG1, TipoG2>(clave: TipoG1, nombre: TipoG2): void {

    console.log(typeof (clave) + ", " + typeof (nombre))

}

mostrarTipo<number, string>(1, "Steve"); // number, string

mostrarTipo<number, string>(1, 2);

// ERROR, argumento 2 no es un string

### Genéricos con restricciones

Si para un genérico queremos limitar solo a algunos tipos concretos, aquellos que podemos sustituir por el alias, debemos utilizar un interfaz y, después, indicar que el genérico extiende la interfaz que hemos creado.

Ejemplo**:**

interface tieneLength {

    length: number;

}

function longitud<Tipo1 extends tieneLength>(x: Tipo1) {

    console.log(x.length);

}

longitud(3);

//Error, argumento 3 no se puede asignar a parámetro

//con length

longitud("Hola"); //4

longitud({ length: 3, width: 5 });

//3 (podemos pasar un objeto que tenga una

//propiedad length tipo number)

longitud([3, 5, 2, 4]); //4

# FUNCIONES

La principal diferencia entre la declaración de las funciones en *TypeScript* y lo que ya conocemos en *JavaScript* es que se introduce el tipado de los parámetros y, también, del dato de retorno de la función.

Si pasáramos un parámetro de otro tipo o retornásemos un parámetro de tipo distinto al indicado, se produciría un error en compilación. En el caso de que la función no retornara nada, podríamos marcarlo como ***void***. En este caso el ***return*** de la función sólo podría devolver ***null*** o ***undefined***.

Sintaxis TypeScript:

//Sintaxis Typecript:

function nombreFuncion(parametro1: tipo1.Parametro2: tipo2): tipoRetorno {

    Bloque;

    [return valor];

}

//Sintaxis JavaScript:

function nombreFuncion(parametros) {

    Bloque;

    [return valor];

}

Ejemplo TypeScript:

    function suma(x: number, y: number): number {

        let total: number = x + y;

        return total;

    }

    console.log(suma(89, 12)); //101

    Ejemplo JavaScript:

    function suma(x, y) {

        var total = x + y;

        return total;

    }

    console.log(suma(89, 12)); //101

A pesar de ser *TypeScript* de tipado fuerte, en muchas ocasiones los tipos pueden ser inferidos, tal y como hemos visto en la nota técnica dedicada a tipos de variables.

    function inicializa(){return 0; }

    // infiere que el tipo de retorno es number

    console.log(inicializa()); //0

## Funciones anónimas

La sintaxis es casi idéntica a *JavaScript*, de nuevo la única diferencia es el tipado.

Ejemplo TypeScript:

   var suma = function (x: number, y: number): number {

        return x + y;

   }

   var resultado: number = suma(3, 5);

   console.log(resultado); //8

Ejemplo JavaScript:

    var suma = function (x, y) {

        return x + y;

    };

    var resultado = suma(3, 5);

    console.log(resultado); //8

## Funciones flecha

También llamadas funciones arrow o lambda, ya vistas anteriormente en un tema específico en Javascript. En TypeScript siguen el mismo formato visto, pero especificando los tipos de los parámetros y valor de retorno.

Usando la flecha **=>**, eliminamos la necesidad de usar la palabra clave **function**. Los parámetros se pasan entre paréntesis () y la expresión de la función se incluye entre corchetes { }.

Ejemplo**:**

    // En Typescript

    let suma = (x: number, y: number): number => {

        return x + y;

    }

    console.log(suma(10, 25)); // 35

    //En Javascript:

    var suma = function (x, y) {

        return x + y;

    };

    console.log(suma(10, 25)); // 35

## Métodos

Cuando declaramos una función dentro de una clase, lo que conocemos como un método, ya no se utiliza la palabra reservada function.

    class Ficha {

        nombre: string;

        edad: number;

        constructor(a: string, b: number) {

            this.nombre = a;

            this.edad = b;

        }

        getEdad(): number {

            return this.edad;

        }

    }

//Instanciamos un objeto de tipo Ficha

    let s = new Ficha("Juan", 22);

    console.log(s.getEdad()); //22

## PARÁMETROS

Del mismo modo que hacíamos en *JS*, en *TypeScript* también podemos pasar parámetros por defecto a nuestras funciones. Al igual que están disponibles los operadores ***rest*** y ***spread***, el uso de funciones es análogo al que vimos para *JS* añadiendo el tipado, por lo que veremos simplemente un ejemplo en cada caso, prestaremos más atención a las novedades respecto *JavaScript*.

### Spread

Cree una copia superficial de los objetos:

    const names = { cat: 'Bob', dog: 'Fred', alligator: 'Benedict' };

    const newNames = { ...names };

    console.log(newNames.cat); // Bob

Puede fusionar varios objetos en un nuevo objeto:

    const names1 = { cat: 'Bob' };

    const names2 = { dog: 'Fred', alligator: 'Benedict' };

    const newNames = { ...names1, ...names2 };

    console.log(newNames.alligator); // Benedict

También puede agregar nuevas propiedades, además de usar una extensión:

    const names = { cat: 'Bob', dog: 'Fred', alligator: 'Benedict' };

    const newNames = { ...names, hamster: 'Jude' };

    console.log(newNames.hamster); // Jude

Tenga en cuenta que si una propiedad se inserta varias veces, la última en gana:

    const names = { cat: 'Bob', dog: 'Fred', alligator: 'Benedict' };

    const newNames = { dog: 'Skip', ...names };

    console.log(newNames.dog); // Fred

Ejemplo2 spread:

    let tabla = [31, 15, 17, 23, 19];

    let minimo =Math.min(...tabla);

    console.log(minimo); // 15

### Rest

Similar a lo que ya podemos hacer en JavaScript puro con la [desestructuración de objetos](https://www.digitalocean.com/community/tutorials/js-object-array-destructuring-es2015) ,descanso nos trae algo que está en proceso para ES2017:

    const names = { cat: 'Bob', dog: 'Fred', alligator: 'Benedict' };

    const { cat, ...otherAnimals } = names;

    console.log(cat); // Bob

    console.log(otherAnimals); // {dog: 'Fred', alligator: 'Benedict'}

Ejemplo2 rest:

    function suma(...lista: number[]) {

        let resultado = 0;

        for (let i = 0; i < lista.length; i++)

            resultado += lista[i];

        return resultado;

    }

    console.log(suma(7, 8)); //15

    console.log(suma(7, 8, 3)); //18

### Parámetros por defecto

En la declaración de la función inicializamos los parámetros por defecto.

Ejemplo:

    function suma(x:number, y:number, z:number=1): number {

        return x + y + z;

    }

    console.log(suma(7,8)); //16

### Parámetros opcionales

En *TypeScript* podemos indicar que un parámetro es opcional agregando el carácter interrogación, **?**, **al final** del nombre del parámetro opcional.

Ejemplo**:**

    function suma(x: number, y: number, z?: number): number {

        if (z) return x + y + z;

        else return x + y;

    }

    console.log(suma(7, 8)); //15

    console.log(suma(7, 8, 3)); //18

Debemos tener la precaución de declarar los parámetros opcionales siempre **al final** de la lista de parámetros pasados a la función.

**5. Parámetros**

## Funciones como parámetros

Se llaman funciones *callback* a aquellas funciones que se pasan como parámetro de otra y se invocan desde dentro de esta.

Ejemplo:

function suma(a: number, b: number): number {

    return a + b;

}

var calculo = function operacion(

a: number, b: number, f: (c: number, d: number) => number): number {

     return f(a, b);

}

console.log(calculo(21, 4, suma)); //25

### Unión de Tipos

Al indicar el tipo de un parámetro de una función, puede ocurrirnos que este pueda ser de dos tipos distintos dependiendo de dónde se invoque. Esto lo solucionamos con la unión de tipos, del mismo modo que vimos en el módulo de variables.

Ejemplo:

function suma(x: number | string, y: number | string): number | string {

    if (typeof x === 'number' && typeof y === 'number')

        return x + y;

    else

        return x.toString() + y.toString();

}

console.log(suma(21, 4)); //25

console.log(suma('hola ', 4)); //hola 4

## Sobrecarga de Funciones

La sobrecarga de funciones (Function overloading) nos permite crear varias funciones con un mismo nombre, pero con diferentes parámetros (diferente número de parámetros o diferente tipo). La implementación después es única y deberemos considerar todas las sobrecargas que hemos hecho. Esto es una novedad respecto a JavaScript.

Ejemplo:

    function suma(x: number, y: number): number;

    function suma(x: number, y: number, z: number): number;

    function suma(x: number, y: number, z?: number): number {

        if (z) return x + y + z;

        else return x + y;

    }

    let n = suma(2, 2);

    console.log(n); // 4

    n = suma(2, 2, 4);

    console.log(n); //8

Ejemplo:

    function divide(a: number, b: number): number;

    function divide(a: string, b: number): string[];

    function divide(x: any, y: number): any {

        if (typeof x == 'number') {

            return x / y;

        } else if (typeof x == 'string') {

            return [x.substring(0, y), x.substring(y)];

        }

    }

    let n: number = divide(10, 2);

    console.log(n); //5

    let s: string[] = divide("supermercado", 5);

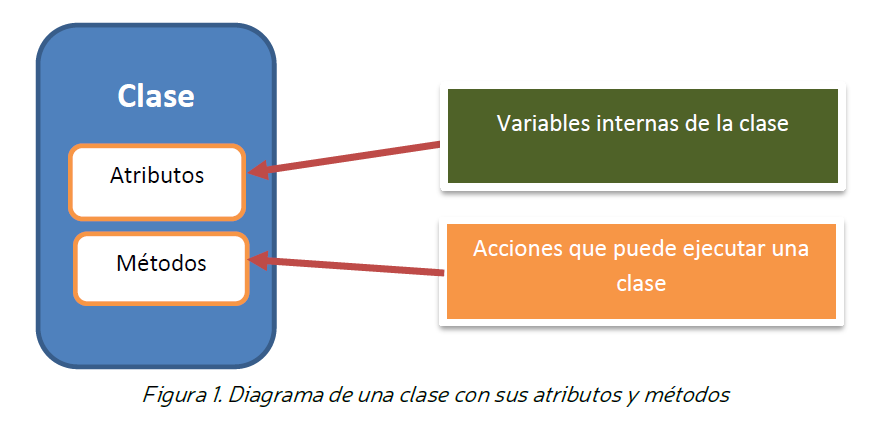
    console.log(s); //Array [‘super’, ‘mercado’]

# POO

## CLASES

Una Clase es un modelo que define un conjunto de atributos (variables) y métodos (funciones) apropiados para operar con dichos datos.

las clases se nombran con la primera letra en mayúsculas (Upper CamelCase), y los objetos con minúsculas (lower CamelCase).



La Clase define una plantilla con las características (atributos y métodos) que deben cumplir los objetos que pertenecen a ella.

En TypeScript definimos una clase con la palabra reservada **class**, y encerramos el cuerpo de la misma entre llaves **{}**. El cuerpo es el espacio dónde definiremos atributos y métodos.

    class NombreClase {

        cuerpo;

    };

• Se utiliza la palabra reservada **class**, para indicar que estamos definiendo una clase.

• Una clase no tiene argumentos, por eso no lleva paréntesis en su definición.

• El punto y coma final en la definición no es necesario.

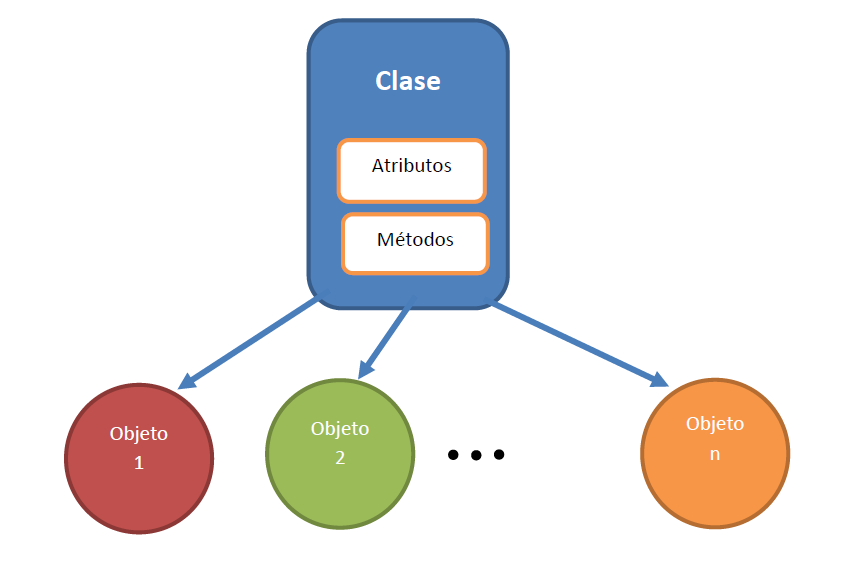
• Se han de declarar las clases previamente a instanciarlas.

• Podríamos considerar una clase como una constante, ya que no se puede modificar la declaración dentro del mismo ámbito.

• Para poder referenciar una clase sólo se puede hacer desde un constructor, por tanto, no tiene un valor de retorno como si fuera una función.

## OBJETOS

Un objeto es una entidad que agrupa un estado y una funcionalidad relacionada, a partir de una determinada plantilla definida por una Clase. Cada objeto almacena y contiene su propio estado interno (atributos), de forma independiente de los otros objetos de la clase.



Un objeto es una entidad dinámica, se crea y se destruye durante la ejecución del programa.

El objeto podrá ser manipulado e interactuar con otros objetos a través de los métodos definidos por la Clase a la que pertenece.

Otra forma de ver un objeto es considerarlo como una colección de propiedades, entendiendo por propiedad las parejas clave/valor. Si la propiedad es un dato, del tipo que sea, la llamaríamos atributo, y si el valor es una función, en lugar de propiedad lo llamaríamos método.

En *TypeScript*, para crear un objeto usamos la palabra reservada **new**, que llamará al constructor de la clase definido en el cuerpo de la misma.

    class NombreClase {

        cuerpo de la clase;

    }

    var nombreObjeto1 = new NombreClase(Parametros);

También tendremos que declarar el tipo de variable para el objeto que vamos a crear, utilizando una de las tres opciones **var**, **let** o **const**, habituales en el uso del lenguaje *JavaScript*.

## ATRIBUTOS

Los atributos son variables internas de la clase que almacenan los valores del estado interno del objeto. Cada objeto tiene su propio estado interno asociado (unos determinados valores para sus atributos), independiente de los otros objetos. Al conjunto de los atributos de una clase se les llama estado de la clase.

En ocasiones también se les denomina propiedades o variables del objeto. Los atributos (variables internas) pueden ser cualquier tipo de *TypeScript* (*string*, *number*, *boolean*, etc) o pueden ser, también, otros objetos.

La definición irá en el cuerpo de la clase, el formato para declarar un atributo debe incluir el nombre del mismo seguido de ‘**:**’ y a continuación el tipo:

    class NombreClase {

        nombreAtributo: TipoAtributo;

    }

Ejemplo:

    class Rueda {

        marca: string;

        modelo: string;

        medida: number;

    }

    class Coche {

        marca: string;

        modelo: string;

        itvVigor: boolean;

        ruedas: Rueda[];

    }

Se han definido dos clases:

• La clase Rueda con dos atributos tipo *string* (marca, y modelo) y un atributo *number* (medida).

• La clase Coche tiene dos atributos de tipo *string* (marca y modelo), un atributo *boolean* (itvVigor), y un atributo (ruedas) que es un *array* de objeto de la clase Rueda.

Vemos como no hay conflicto en el nombre de los atributos, aunque se repitan, ya que pertenecen a clases distintas.

Es buena práctica poner un guion bajo antes del nombre del atributo para diferenciarlos de los *getters* y los *setters*.

    marca: \_string;

    modelo: \_string;

## MÉTODOS

Como hemos visto, una clase representa una abstracción de datos, y los métodos definen el comportamiento sobre estos datos.

Los métodos son algoritmos especiales que son definidos por la clase, y que se aplican directamente sobre los objetos. Actúan sobre el estado interno (atributos) del objeto sobre el que se aplican. También se les denomina métodos de instancia, o métodos del objeto.

A efectos prácticos un método no es más que una función dentro de una clase.

La declaración de un método en *TypeScript*, se realiza dentro del cuerpo de la clase, y de la forma que sigue:

    class NombreClase {

        nombreMetodo(Parámetros): TipoRetorno {

            cuerpo del método;

        }

    }

Si no retorna valor, se omite el tipo de retorno, de la forma:

    class NombreClase {

        nombreMetodo(Parámetros) {

            cuerpo del método;

        }

    }

Para llamar a un método se hace de forma análoga a un atributo, pero se le añade al final dos paréntesis **()**, dentro de los cuales irían los parámetros que le pasamos al método en caso de que los hubiera.

Para poder invocar un método desde fuera del objeto (fuera de su contexto), se haría llamando en primer lugar al objeto, y utilizando el separador punto invocar al método del objeto seguido de los paréntesis que hemos comentado. Es decir, la nomenclatura del punto.

    nombreObjeto.nombreMetodo(Parámetros);

Existen algunos tipos especiales de métodos: constructores, destructores, accesores,…

### Métodos constructores

En todas las clases debemos crear un método especial “constructor”, método que crea un objeto de esta clase e inicializa los atributos de la misma. Normalmente el nombre del constructor es el mismo que el de la clase, y no devuelve ningún valor. Sólo puede haber un constructor por clase.

Siempre se debe declarar la clase antes de crear el objeto.

Todos los constructores se definen usando la palabra reservada ***constructor***.

    class NombreClase {

        constructor(Parámetros) {

            /\* Código del constructor \*/

        }

    }

    var nombreObjeto1 = new NombreClase(Parámetros);

Todas las clases se crean con un constructor por defecto. Si no definimos ningún constructor, *TypeScript* creará este constructor, que no tiene parámetros y no hace ninguna acción.

Al crear un objeto de una clase, estamos llamando al método constructor de la misma, por tanto, la creación del objeto siempre ha de ser posterior a la declaración de la clase de este.

Ejemplo**:**

En el siguiente ejemplo se produce un error por intentar crear un objeto de tipo Coche antes de declarar la clase.

    var coche0 = new Coche("Volkswagen", "Passat");

//error TS2449: La clase coche aun no ha sido declarada

    class Coche {

        marca: string;

        modelo: string;

        constructor(a: string, b: string) {

            this.marca = a;

            this.modelo = b;

        }

    }

Ejemplo:

Lo correcto es primero declarar la clase, y después crear el objeto. Primero se crea la clase Coche y después se utiliza.

    class Coche {

        \_marca: string;

        \_modelo: string;

        constructor(a: string, b: string) {

            this.\_marca = a;

            this.\_modelo = b;

        }

    }

    var coche1 = new Coche("Seat", "Altea");

    console.log(coche1.marca); //Seat

### This

*TypeScript* dispone de la palabra reservada **this**, para referirse al objeto actual que invocó el método. A partir del **this** podemos acceder a las propiedades y métodos del objeto. Vemos su funcionamiento en el ejemplo anterior.

*TypeScript* dispone de una forma simplificada de crear e inicializar al mismo tiempo los atributos de un constructor.

Ejemplo: el siguiente código es equivalente al anterior.

    class Coche {

        constructor(

            public \_marca: string,

            public \_modelo: string

        ) { }

    }

    var coche1 = new Coche("Seat", "Altea");

    console.log(coche1.\_marca); //Seat

### Destructores

El destructor es un método, al que se llama automáticamente en la ejecución de un programa, cuando se llega al final del ámbito donde se declaró el objeto. Se emplea para liberar esos recursos obtenidos en el constructor y eliminar los vínculos a las posiciones de memoria.

**5. Métodos**

### Métodos accesores (Getter y Setters)

Los métodos accesores son simples funciones que definimos en las clases para mostrar (***get***) o modificar (***set***) el valor de un atributo. Para distinguirlos se utilizan las palabras "***get***" o "***set***" como prefijos, seguidos del nombre de la propiedad a la que queremos acceder, por ejemplo: getMarca().

Hay que tener la precaución de no entrar en referencias circulares. Si se utiliza el mismo nombre para el método ***set*** que para la propiedad a la que vamos a asignarle el valor, se puede llegar a error (RangeError: Maximum call stack size exceed): estaríamos llamando al método ***set*** desde el cuerpo de sí mismo.

Por ello, se suele seguir el convenio de anteponer un guion bajo al nombre del atributo para que este no coincida exactamente con el nombre del método.

Aunque no hay restricciones, más allá de que no coincidan los nombres utilizados, como ya se ha comentado.

Ejemplo**:**

En el siguiente ejemplo declaramos los atributos para los que se van a definir los métodos accesores iniciando con “**\_**” y definimos las funciones ***get*** y ***set*** para cada uno de los atributos.

    class Coche {

        \_marca: string;

        \_modelo: string;

        constructor(a: string, b: string) {

            this.\_marca = a;

            this.\_modelo = b;

        }

        set marca(a: string) {

            this.\_marca = a;

        }

        set modelo(b: string) {

            this.\_modelo = b;

        }

        get marca() {

            return this.\_marca;

        }

        get modelo() {

            return this.\_modelo;

        }

    }

    var coche1 = new Coche("Seat", "Altea");

    coche1.marca = "Opel";

    console.log(coche1.marca); //Opel

**Nota**: Puede que al intentar compilar el ejemplo anterior aparezca el error: “error TS1056: Accessors are only available when targeting ECMAScript 5 and higher.”.

Esto es debido a que no estamos compilando en ES6. Para forzar esta situación, utilizaremos en la compilación la opción “es6”

tsc -t es6 <nombre del fichero.ts>

#### getters y setters implícitos

*TypeScript* permite hacerlo también de una manera mucho más sencilla, accediendo directamente al atributo mediante el separador punto “**.**”, utilizándolo en la parte izquierda de una igualdad haríamos una asignación o modificación de este atributo. Si se utiliza en la parte derecha, se trataría de un ***get***, y nos devolvería el valor del atributo. No es lo más utilizado, ya que, como veremos en la nota técnica de encapsulamiento, se recomienda no acceder directamente a los atributos de la clase. Es mejor definirlos como atributos privados y acceder a dichos atributos mediante ***getters*** y ***setters***.

En el siguiente ejemplo se muestra el uso de ***getter*** y ***setter*** implícitos en la clase Coche de los ejemplos anteriores:

    class Coche {

        marca: string;

        modelo: string;

        constructor() {

        }

    }

    var coche1 = new Coche();

    coche1.marca = "Opel";

    coche1.modelo = "Astra";

    console.log(coche1.marca); //Opel

    console.log(coche1.modelo); //Astra

**5. Métodos**

### Métodos Personalizados

Además de los métodos especiales se podrán crear los métodos necesarios que más se adecúen a nuestro propósito para definir el comportamiento de una clase.

Los métodos personalizados serán los encargados de implementar el funcionamiento de una clase, definiendo las acciones que se pueden llevar a cabo desde un objeto.

Ejemplo:

En el siguiente ejemplo, creamos un método personalizado para que nos devuelva la marca y modelo de un Coche:

    class Coche {

        marca: string;

        modelo: string;

        constructor(a: string, b: string) {

            this.marca = a;

            this.modelo = b;

        }

        mostrarDatos(): void {

            console.log(this.marca, this.modelo);

        }

    }

    var coche1 = new Coche("Seat", "Altea");

    coche1.mostrarDatos(); //Seat Altea

Otros ejemplos

Se puede realizar un programa que permita registrar la marca y el modelo de distintos coches, así como la marca, modelo y medida de las ruedas que lleva. Para ello, es necesario poder cambiar una rueda o las cuatro, y mostrar por consola la información de cualquier coche y cualquier rueda. El código sería el siguiente:

class Rueda {

    marca: string;

    modelo: string;

    medida: number;

    constructor(a: string, b: string, c: number) {

        this.marca = a;

        this.modelo = b;

        this.medida = c;

    }

}

class Coche {

    marca: string;

    modelo: string;

    ruedas: Rueda[];

    constructor(a: string, b: string, d: Rueda[]) {

        this.marca = a;

        this.modelo = b;

        this.ruedas = [];

    }

    mostrarDatos(): void {

        console.log(this.marca, this.modelo);

    }

    cambiarRuedas(a: string, b: string, c: number) {

        var r = new Rueda(a, b, c);

        this.ruedas = [r, r, r, r];

    }

    cambiar1Rueda(a: string, b: string, c: number, d: number) {

        this.ruedas[d] = new Rueda(a, b, c);

    }

    mostrarRueda(a: number): void {

console.log(this.ruedas[a].marca, this.ruedas[a].modelo,

this.ruedas[a].medida);

    }

}

var coche1 = new Coche("Seat", "Altea", []);

var coche2 = new Coche("Audi", "A4", []);

coche1.cambiarRuedas("Michelin", "Pilot", 17);

coche1.mostrarDatos();//Seat Altea

coche1.mostrarRueda(3);//Michelin Pilot 17

coche1.cambiar1Rueda("Dunlop", "Sport", 17, 3);

coche1.mostrarRueda(3);//Dunlop Sport 17

## Atributos y Métodos estáticos.

En ocasiones los objetos de una misma clase necesitan compartir información. Para ello usaremos atributo o propiedades estáticos (también llamados de clase). Estos elementos no pertenecen al objeto (*this*) sino a la clase.

En estos casos en lugar de acceder al objeto tenemos que acceder a la clase:



Esto también significa que se creará sólo una copia de la variable.

De igual forma también tenemos los métodos de clase, para los cuales no necesitamos instanciar un objeto de clase.



## STATIC

Las clases pueden tener elementos **static**. Estos miembros no están asociados con una instancia particular de la clase. Se puede acceder a ellos a través del propio objeto constructor de la clase:

    class MyClass {

        static x = 0;

        static printX() {

            console.log(MyClass.x);

        }

    }

    console.log(MyClass.x);

    MyClass.printX();

Probar

Los miembros estáticos también pueden utilizar las mismas public, protectedy privatede visibilidad modificadores:

    class MyClass {

        private static x = 0;

    }

    console.log(MyClass.x);

    Property 'x' is private and only accessible within class 'MyClass'.

Probar

Los miembros estáticos también se heredan:

    class Base {

        static getGreeting() {

            return "Hello world";

        }

    }

    class Derived extends Base {

        myGreeting = Derived.getGreeting();

    }

Probar

### Nombres estáticos especiales

Por lo general, no es seguro ni posible sobrescribir las propiedades del Functionprototipo. Debido a que las clases son funciones en sí mismas con las que se puede invocar new, staticno se pueden usar ciertos nombres. Las propiedades de la función como name, lengthy callno son válidas para definir como staticmiembros:

    class S {

        static name = "S!";

        //Static property 'name' conflicts with built - in property 'Function.name' of constructor function 'S'

    }

Probar

### ¿Por qué no hay clases estáticas?

TypeScript (y JavaScript) no tienen una construcción llamada de static classla misma manera que C # y Java.

Esas construcciones solo existen porque esos lenguajes obligan a todos los datos y funciones a estar dentro de una clase; debido a que esa restricción no existe en TypeScript, no es necesario. Una clase con una sola instancia generalmente se representa como un objeto normal en JavaScript / TypeScript.

Por ejemplo, no necesitamos una sintaxis de "clase estática" en TypeScript porque un objeto normal (o incluso una función de nivel superior) hará el trabajo igual de bien:

    // Unnecessary "static" class

    class MyStaticClass {

        static doSomething() { }

    }

    // Preferred (alternative 1)

    function doSomething() { }

    // Preferred (alternative 2)

    const MyHelperObject = {

        dosomething() { },

    };

# INTERFACES

Una interface es una plantilla vacía que los objetos van a tener que cumplir para poder implementarla, es un esqueleto.

Es un conjunto de métodos abstractos y de constantes cuya funcionalidad es determinar el funcionamiento de una clase.

Al ser sus métodos abstractos, por lo que no tienen funcionalidad alguna. Sólo se definen el tipo, el argumento y el tipo de retorno que va a tener.

Son un tipo de construcción utilizada en la [Programación Orientada a Objetos](https://desarrolloweb.com/manuales/teoria-programacion-orientada-objetos.html) con la que tratamos de suplir la carencia de herencia múltiple.

La mayoría de los lenguajes que implementan la orientación a objetos no ofrecen la posibilidad de definir una clase que extienda varias clases a la vez y sin embargo a veces es deseable. Ahí es donde entran las interfaces.

Una clase puede extender otra clase, heredando sus propiedades y métodos y declarar que implementa cualquier número de interfaces. La diferencia de las clases que extiendes con respecto a las interfaces es que las interfaces no contienen implementación de sus métodos, por lo que la clase que implementa una interfaz debe escribir el código de todos los métodos que contiene. Por este motivo, se dice que las interfaces son como un contrato, en el que se especifica las cosas que debe contener una clase para que pueda implementar una interfaz o cumplir el contrato declarado por esa interfaz.

Ese sería el concepto de manera genérica. Luego cada lenguaje puede tener ligeras diferencias a la hora de aplicar interfaces. Por ejemplo, en TypeScript una interfaz puede definir propiedades, mientras que en otros lenguajes las interfaces sólo definen métodos.

## Declarar una interfaz en TypeScript

Las interfaces en TypeScript se declaran de manera bastante similar a la de las clases, indicando la lista de propiedades y métodos que contendrán. Solo hay un detalle fundamental, que las propiedades no pueden tener valores y los métodos no pueden tener código para su implementación ya que son abstractos, por lo que lal implementación la harán las clases que implementen la interface.

Aquí podemos ver el código de una interface llamada "sumergibleInterface".

    interface sumergibleInterface {

        tiempoMaxBajoElAgua: number;

        profundidadMaxima: number;

        repelerAgua(): void;

    }

Como ves, solo hemos indicado los tipos y los métodos, pero no hemos indicado sus valores. Hemos usado además el tipado de TypeScript, ya que es básico para aprovecharnos de las características del lenguaje.

## Implementar una interfaz

A nivel de programación tradicional, lo más típico que puedes hacer con una interfaz es implementarla en una clase.

Una vez definida tu interfaz, podrás implementarla en todas las clases que desees mediante la palabra "implements" en la cabecera de la clase. Para nuestro ejemplo, una vez definida la interfaz sumergibleInterface, todas las clases que vayan a tener objetos que se puedan sumergir, tendrán que implementarla.

    class relojSumergible implements sumergibleInterface {

        tiempoMaxBajoElAgua = 1;

        profundidadMaxima = 10;

        repelerAgua() {

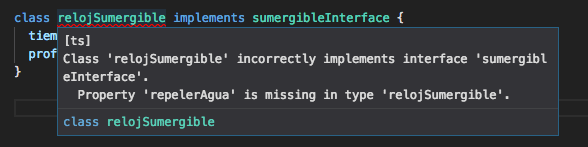
            console.log('El agua me resbala');

        }

    }

La interfaz provoca que sea necesario declarar todas las propiedades e implementar todos los métodos a la hora de definir la clase. En resumen, es como un contrato.

Si no se cumple el contrato de la interfaz, entonces nuestro editor se quejará (si está preparado para mostrar los errores de código TypeScript), o el compilador nos lo advertirá.



Si una clase implementa varias interfaces tiene que implementar todos los métodos de ambas.

interface miinterface1 {

    miMetodo1(): string;

}

interface miinterface2 {

    miMetodo2(): string;

}

class miclass1 implements miinterface1, miinterface2 {

    miMetodo1() {

        console.log('Esta es la implementacio del método1');

    };

    miMetodo2() {

        console.log('Esta es la implementacio del método2');

    };

}

## La interfaz como un nuevo tipo

Pero además, *TypeScript* nos ofrece una aplicación adicional de las interfaces: la creación de un nuevo tipo que podemos usar a lo largo de nuestro código.

    interface citaCalendario {

        fechaHora: Date;

        titulo: string;

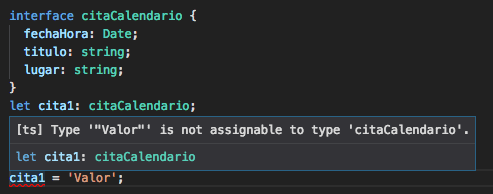
        lugar: string;

    }

    let cita1: citaCalendario;

Como puedes ver, a la hora de crear variables, puedo decir que su tipo es una interfaz declarada. Quizás en un primer momento no le encuentres mucha utilidad, pero realmente es interesante por todas las ayudas que te ofrece el editor, o el compilador de *TypeScript*, a la hora de programar.

Por ejemplo, esto es lo que pasaría si asignas un valor a la variable "cita1" que no cumpla el contrato de la interfaz, por asignarle cualquier otra cosa que no tenga este tipo.



No solo el editor nos advierte, también el compilador de TypeScript.



Para que nadie se queje, necesitarás asignar un valor a la variable "cita1" que concuerde con la interfaz, es decir, cumplir el contrato establecido en la declaración del tipo.

    cita1 = {

        fechaHora: new Date(Date.now()),

        titulo: 'Programar en TypeScript',

        lugar: 'Oficina de DesarrolloWeb.com'

    }

Ahora sí tenemos todas las propiedades con los tipos declarados, por lo que el editor ya no nos marcará esta asignación como un error y el compilador podrá hacer su trabajo de transpilación a *Javascript*.

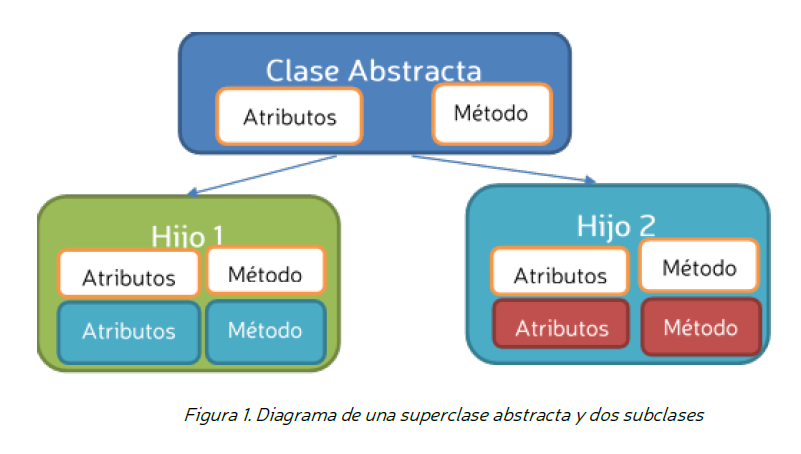
**Conclusión**

Como has podido ver, las interfaces no son complicadas de definir. En el mundo de la programación orientada a objetos se usan bastante, pero lo que queríamos mostrar es que se pueden usar en *TypeScript* para producir ayudas en tiempo de desarrollo.

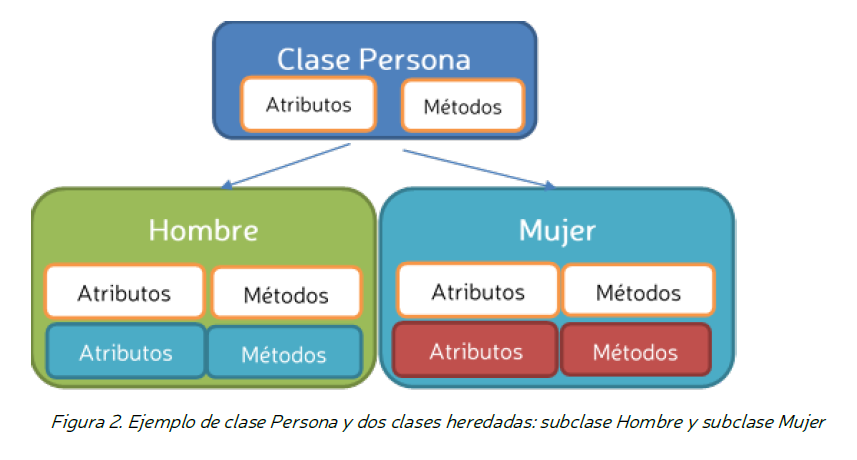
Nos obligan a escribir un poco más de código pero nos ofrecen tipos valiosos, que seremos capaces de usar en nuestras variables. A partir de entonces, cualquier pequeño error a la hora de escribir el código, ya sea porque nos olvidemos de algo, o hagamos un tipeo incorrecto y escribamos mal una propiedad, el editor o el compilador nos lo advertirán, evitando que nos tengamos que volver locos para encontrar errores, que a veces son difíciles de detectar.

# CLASES ABSTRACTAS

Obligan a definir el comportamiento completo de los objetos que hereden de dicha clase.



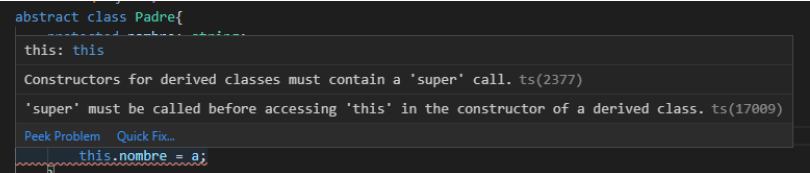
Una clase abstracta no puede ser instanciada, como la clase Persona. Su función es que hereden otras clases de ella, que sí serían instanciables como, por ejemplo, las Clases Hombre y Mujer que se muestran en la figura 2.



*TypeScript* permite el uso de clases abstractas. No se pueden crear objetos directamente de estas clases, sino que es necesario crear previamente otras clases que hereden de estas abstractas, definan todos las propiedades y métodos abstractos declarados en la función padre abstracta. A partir de esas clases ya podremos crear objetos.

La sintaxis consiste en utilizar la palabra reservada **abstract** en la declaración de la clase, previamente a la palabra **class**. En el cuerpo de la clase, las propiedades o métodos abstractos también se pueden declarar con la palabra reservada **abstract** para hacer más legible el código, pero hay que tener en cuenta que los métodos definidos como **abstract** en la clase abstracta no podrán incluir implementación en esta clase y deberán ser implementados en las clases hijas.

Habitualmente, no queremos repetir el código del mismo comportamiento, que se repite entre clases padres e hijas. Para ahorrar código y reutilizar el que se ha escrito ya en una clase padre, las clases hijas pueden invocar desde su constructor al constructor de la clase padre, con la palabra reservada ***super()***. El editor nos dará indicaciones en este sentido, como se muestra en el ejemplo siguiente:



Ejemplo:

    abstract class Padre {

        abstract comunicar(): void;

    }

    class Hijo extends Padre {

        comunicar(): void {

            console.log("Comunicar en hijo")

        }

    }

    class Hija extends Padre {

        comunicar(): void {

            console.log("Comunicar en hija");

        }

    }

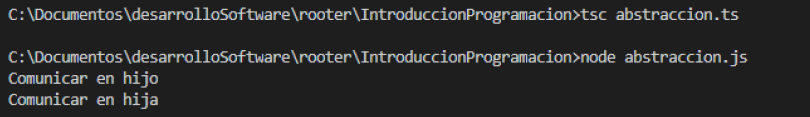
    let hijo: Hijo = new Hijo();

    let hija: Hija = new Hija();

    hijo.comunicar();//Nada en hijo

    hija.comunicar();//Nada en hija

La salida esperada en la consola sería la siguiente:



Ejemplo:

abstract class Ficha {

        abstract nombre: string;

        mostrarNombre(): void {

            console.log(this.nombre);

        }

    }

    class Alumno extends Ficha {

        nombre: string;

        edad: number;

        constructor(a: string, b: number) {

            super();

            this.nombre = a;

            this.edad = b;

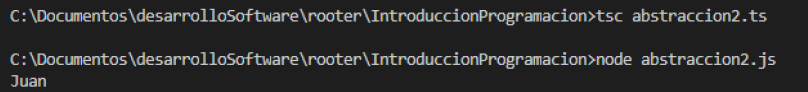
        }

    }

    let a: Ficha = new Alumno("Juan", 17);

    a.mostrarNombre(); //Juan

La salida esperada en este caso es:



Ejemplo:

        abstract class Vehiculo {

        constructor(public \_marca: string, public \_modelo: string) {

            this.\_marca = \_marca;

            this.\_modelo = \_modelo;

        }

        encender(): void {

            console.log("Encendido");

        }

        apagar(): void {

            console.log("Apagado");

        }

        abstract acelerar(): void;

        abstract frenar(): void;

    }

    class CocheVeloz extends Vehiculo {

        constructor(a: string, b: string) {

            super(a, b);

            this.\_marca = a;

            this.\_modelo = b;

        }

        acelerar(): void {

            console.log("Acelerar rápido");

        }

        frenar(): void {

            console.log("Frenar rápido");

        }

    }

    class CocheLento extends Vehiculo {

        constructor(a: string, b: string) {

            super(a, b);

            this.\_marca = a;

            this.\_modelo = b;

        }

        encender(): void {

            console.log("Encendido lento");

        }

        apagar(): void {

            console.log("Apagado lento");

        }

        acelerar(): void {

            console.log("Acelerar lento");

        }

        frenar(): void {

            console.log("Frenar lento");

        }

    }

    let coche1: CocheVeloz = new CocheVeloz('Tesla', 'Model 3');

    let coche2: CocheLento = new CocheLento('Volkswagen', 'Polo');

    coche1.encender();

    coche1.apagar();

    coche1.acelerar();

    coche1.frenar();

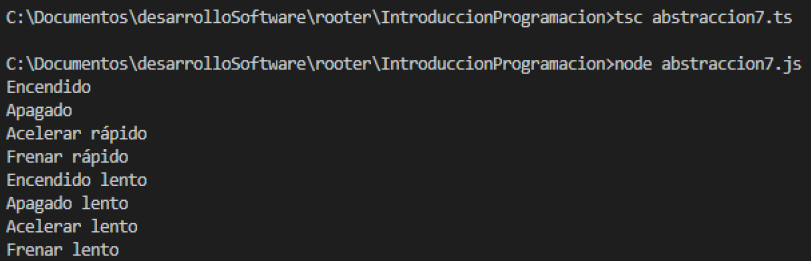
    coche2.encender();

    coche2.apagar();

    coche2.acelerar();

    coche2.frenar();

La salida esperada en la consola es:



# MODIFICADORES DE ACCESO.

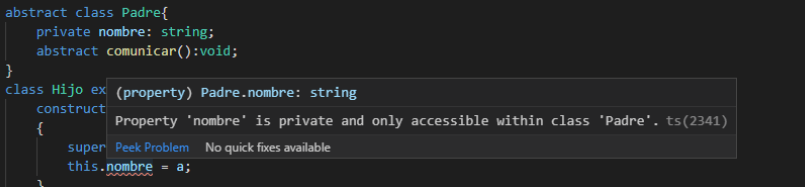
Usando las palabras reservadas ***protected***, ***private*** y ***public*** (ámbito por defecto) podemos definir propiedades y métodos: protegidos, privados y públicos.

Los métodos declarados como ***private*** son solo accesibles desde la propia clase, los declarados como ***protected*** son accesibles desde su clase y clases hijas. Los métodos y atributos declarados como ***public*** serán públicos en todas las partes del programa.

Si no ponemos modificador de acceso se entiende que es ***public***.

Los modificadores de acceso se introducen en varias notas técnicas por lo que se verán también en detalle en otros apartados.

En el caso de las clases abstractas nos interesa el ámbito ***public*** y sobre todo el ***protected***, que será el que permita que las subclases puedan acceder a los atributos y métodos de la superclase. En caso de definirlo como ***private***, las clases que hereden no podrán acceder a dichos atributos/métodos. El editor puede darnos indicaciones en tiempo real sobre esta circunstancia como se ve en la siguiente consola:



Siguiendo con el ejemplo anterior, si definimos el atributo nombre como ***protected***, solamente podrá accederse a él desde las subclases Hijo/Hija. Si intentamos acceder desde el programa principal el compilador mostrará un error, porque no es público:

    abstract class Padre {

        protected nombre: string;

        abstract comunicar(): void;

    }

    class Hijo extends Padre {

        constructor(a: string) {

            super();

            this.nombre = a;

        }

        comunicar(): void {

            console.log("Comunicar en hijo")

            console.log('Mi nombre es: ', this.nombre);

        }

    }

    class Hija extends Padre {

        comunicar(): void {

            console.log("Comunicar en hija");

        }

    }

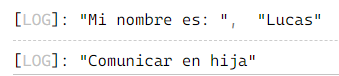
    let hijo: Hijo = new Hijo('Lucas');

    let hija: Hija = new Hija();

    hijo.comunicar();//Nada en hijo

    hija.comunicar();//Nada en hija

En este caso, el resultado de la ejecución en la consola sería el siguiente:



Adicionalmente a los tres modificadores anteriores, podemos utilizar el modificador ***readonly***, para definir propiedades sólo de lectura en una clase.

Ejemplo 2

En este ejemplo se van a comprobar los errores que da el compilador cuando se intenta modificar un atributo ***readonly*** y cuando se intenta acceder a otro atributo que ha sido definido como privado.

El atributo nombre se define como ***readonly***. Al intentar modificar este atributo mediante el método *updateNombre*, el compilador mostrará un error.

El atributo edad se define como privado. Del mismo modo, al intentar acceder a él desde el programa principal, el compilador también mostrará un error dado que no puede accederse a un atributo privado de una clase desde fuera de ella.

**3. Modificadores de acceso**

    class Ficha {

        private readonly nombre: string;

        private edad: number;

        constructor(a: string, b: number) {

            this.nombre = a;

            this.edad = b;

        }

        mostrarDatos(): void {

            console.log(this.nombre, this.edad);

        }

        updateNombre(a: string) {

            this.nombre = a;

        }

        updateEdad(b: number) {

            this.edad = b;

        }

    }

let alumno = new Ficha("Juan", 17);

    alumno.mostrarDatos(); //Juan, 17

    alumno.updateNombre('Pablo');

//error en compilación al ser una propiedad readonly

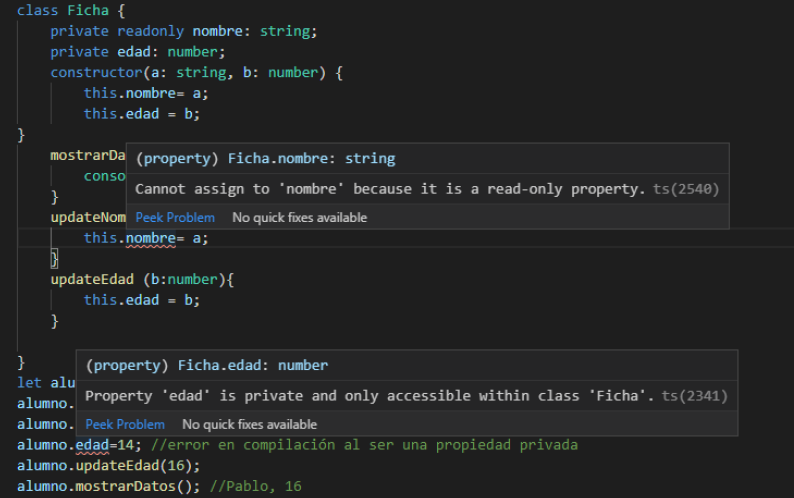
    alumno.edad = 14;

//error en compilación al ser una propiedad privada

    alumno.updateEdad(16);

    alumno.mostrarDatos(); //Pablo, 16

En el editor podremos ver los siguientes avisos:



La propiedad ***readonly*** es parte de los tipos de *TypeScript*, por lo que si se compila a *JavaScript* esta propiedad se pierde. Lo que sí detecta el compilador es que se ha producido un error por intentar asignar una variable ***readonly*** desconocida. Al generar el código *Javascript* ya no se mantiene esta restricción por lo que el resultado no es el esperado, y sí que se puede cambiar el valor del nombre.



El contenido del fichero *Javascript* generado sería el siguiente:

    var Ficha = /\*\* @class \*/ (function () {

        function Ficha(a, b) {

            this.nombre = a;

            this.edad = b;

        }

        Ficha.prototype.mostrarDatos = function () {

            console.log(this.nombre, this.edad);

        };

        Ficha.prototype.updateNombre = function (a) {

            this.nombre = a;

        };

        Ficha.prototype.updateEdad = function (b) {

            this.edad = b;

        };

        return Ficha;

    }());

    var alumno = new Ficha("Juan", 17);

    alumno.mostrarDatos(); //Juan, 17

    alumno.updateNombre('Pablo');

//error al ser una propiedad readonly

    alumno.updateEdad(16);

    alumno.mostrarDatos(); //Juan, 16

Y la salida del programa en la consola:



Si se define la clase *updateNombre* como privada, el compilador también nos informará de error:

    private updateNombre(a: string){

        this.nombre = a;

    }



Pero el código generado en *Javascript*, de nuevo, no incluirá esa restricción. Para evitar que se generen *JavaScript* que puedan tener errores de este tipo, se deberá utilizar la opción “***noEmitOnError***” a la hora de llamar al compilador:

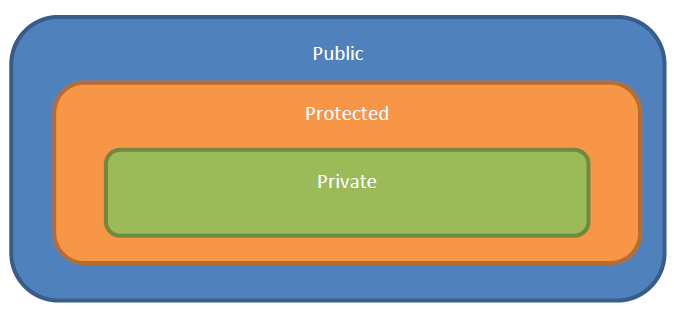
tsc –noEmitOnError <nombrefichero>

Con esta opción no se generará el fichero .js en caso de que contenga errores.

# ENCAPSULACIÓN

Existen tres niveles de acceso definidos para el encapsulamiento: público, protegido y privado, que se comentarán en detalle en los siguientes apartados.

Estos niveles de acceso definirán las normas de acceso a los atributos y métodos de una clase. En el siguiente esquema se muestran del menos restrictivo (public) al más restrictivo (private).



### Público (public)

Los atributos y métodos definidos como públicos pueden ser accedidos desde cualquier punto del programa. Bajo este nivel de acceso se define todo aquello que quiere que vea la parte externa.

Es el nivel de acceso menos restrictivo. Se identifica anteponiendo a la definición del atributo o el método la palabra reservada “**public**”. En muchos lenguajes es implícito y no es necesario identificarlo como “**public**” ya que lo consideran el nivel por defecto.

### Protegido (protected)

Los atributos y métodos definidos como protegidos solamente pueden ser accedidos dentro de su clase y por sus subclases, por lo que bajo este nivel de acceso solo se define todo aquello que quieres acceder dentro de la clase y las clases que hereden de esta clase en la que se define el atributo o método protegido.

Es el nivel de acceso intermedio. Se identifica anteponiendo a la definición del atributo o el método la palabra reservada “**protected**”.

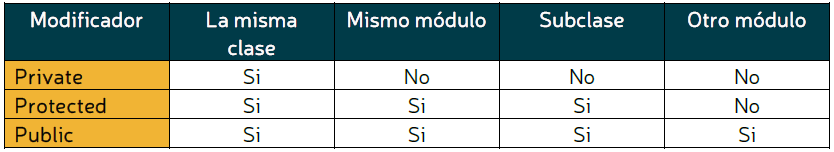
### Privado (private)

Los atributos y métodos definidos como privados solamente pueden ser accedidos desde su propia clase.

Es el nivel de acceso más restrictivo y se identifica anteponiendo a la definición del atributo o el método la palabra reservada “**private**”.

La mejor forma de ver y comprender el uso y el funcionamiento del encapsulamiento es mediante ejemplos. A continuación, se detallarán unos escenarios codificados en TypeScript.

El funcionamiento de las restricciones de los diferentes niveles puede verse en la siguiente tabla:



# HERENCIA

La **herencia** significa que se pueden crear nuevas clases partiendo de clases existentes, que tendrá todas los atributos y los métodos de su 'superclase' o 'clase padre' y además se le podrán añadir otros atributos y métodos propios.

Tendremos una clase padre o “*superclase*” y una o varias clases hijas.

Para la definición de superclases, es muy habitual utilizar clases abstractas en las que se definen los atributos principales y los métodos básicos que se utilizarán en las subclases.

La subclase puede tener sus propios atributos y métodos, además de los atributos y métodos de la superclase.

*Typescript* sólo permite la herencia simple, no se permite la herencia múltiple, es decir, una clase hija sólo puede heredar de una sola clase padre.

En *TypeScript*, para indicar que una clase hereda de otra, se utiliza la palabra “***extends***” en la declaración de la clase, de la siguiente forma:

    class Herbivoro extends Animal {

        // cuerpo de la clase Herbivoro

    }

Los atributos de la supeclase debieran ser siempre privados, y acceder a ellos mediante métodos ***get*** y ***set.***

    abstract class Animal {

        private \_animalName: string;

        private \_patas: number;

        constructor(name: string, patas: number) {

            this.\_animalName = name;

            this.\_patas = patas;

        }

        public get animalName(): string {

            return this.\_animalName;

        }

        public set animalName(value: string) {

            this.\_animalName = value;

        }

        public get patas(): number {

            return this.\_patas;

        }

        public set patas(value: number) {

            this.\_patas = value;

        }

        abstract saludo(): void;

    }

    class Perro extends Animal {

        constructor(name: string, patas: number) {

            super(name, patas);

        }

        saludo() {

            console.log('Guau soy ' + this.animalName);

        }

    }

    class Vaca extends Animal {

        constructor(name: string, patas: number) {

            super(name, patas);

        }

        saludo() {

            console.log('Muuuu soy ' + this.animalName);

        }

    }

    let miPerro1 = new Perro('Rober', 4);

    miPerro1.saludo();

    let miPerro2 = new Perro('Rhin', 4);

    miPerro2.saludo();

    let miVaca1 = new Vaca('Nervi', 4);

    miVaca1.saludo();

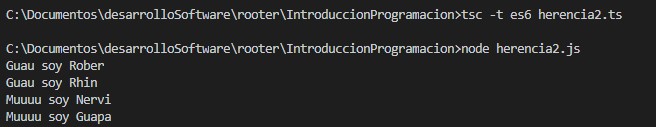
    let miVaca2 = new Vaca('Guapa', 4);

    miVaca2.saludo();

Puesto que el uso de los ***getters*** y ***setters*** están soportados a partir de ECMAScript 5, se precisará indicar al compilador la opción de compilación adecuada:

tsc -t es6 <nombre del fichero.ts>

Sin ningún cambio adicional, la salida del programa codificado anteriormente será:

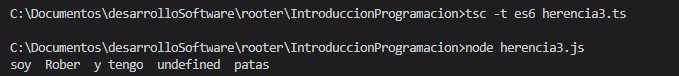


Ejemplo 2: Acceso y modificación de los atributos de la clase padre

Si, por ejemplo, el atributo patas no se inicializase en el constructor de la clase padre, si consultamos dicho atributo:

console.log('soy ', miPerro1.animalName, ' y tengo ', miPerro1.patas, ' patas');

El valor sería ‘undefined’:



Suponiendo que no hubiésemos metido patas en el constructor de la clase animal, dado que el atributo patas, es algo específico de cada Animal, podremos asignar el numero adecuado patas en el constructor de la clase hija.

Modificamos el constructor de las clases Perro y Vaca:

    class Perro extends Animal {

        constructor(name: string) {

            super(name);

this.patas = 4;

        } saludo() { console.log('Guau soy ' + this.animalName); }

    }

    class Vaca extends Animal {

        constructor(name: string) {

            super(name);

this.patas = 4;

        } saludo() { console.log('Muuuu soy ' + this.animalName); }

    }

Si tuviéramos la clase ciempiés, indicaríamos en su constructor patas=100;

Como ya estamos inicializando el dato en el constructor de la clase mediante el uso del set patas (this.patas=4), si consultamos el número de patas de nuestras instancias, veremos que todas ellas nos devuelven el valor 4:

let miPerro1 = new Perro('Rober'); miPerro1.saludo();

let miPerro2 = new Perro('Rhin'); miPerro2.saludo();

let miVaca1 = new Vaca('Nervi'); miVaca1.saludo();

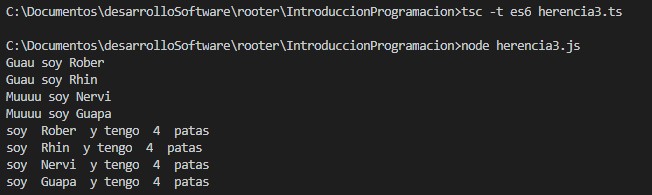
let miVaca2 = new Vaca('Guapa'); miVaca2.saludo();

console.log('soy ',miPerro1.animalName,' y tengo ',miPerro1.patas,' patas');

console.log('soy ',miPerro2.animalName,' y tengo ',miPerro2.patas,' patas');

console.log('soy ',miVaca1.animalName,' y tengo ',miVaca1.patas,' patas');

La salida de programa es:



|  |
| --- |
| class Perro extends Animal { private \_cazador: boolean; constructor (name: string){ super(name); this.patas = 4; this.\_cazador = false;  } public get cazador(): boolean { return this.\_cazador;  } public set cazador(value: boolean) { this.\_cazador= value;  } |

Ejemplo 3: Añadiendo métodos específicos de las clases hijas

Para añadir especialización a las clases hijas, lo que se hace es añadir nuevos atributos, que definirán características propias de la clase, y nuevos métodos, que añadirán comportamiento específico para la clase.

En el caso de la clase Perro vamos a añadir un nuevo atributo que nos indique si el perro es cazador o no. Por defecto, en el constructor de la clase indicaremos que los perros NO son cazadores, por tanto, inicializaremos dicho atributo a false. De modo que, salvo que se indique lo contrario, las instancias de perro creadas por defecto no serán cazadores.

Para facilitar las comprobaciones modificamos la función “saludo” para que nos muestre todos los datos de la instancia.

saludo() {

    console.log('Guau soy ', this.animalName, 'tengo ',

this.patas, ' patas y ', this.cazador ? 'Si' : 'No', ' soy cazador');

}

let miPerro1 = new Perro('Rober');

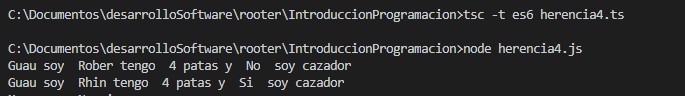
miPerro1.saludo();

let miPerro2 = new Perro('Rhin');

miPerro2.cazador = true;

miPerro2.saludo();

La salida del programa sería:

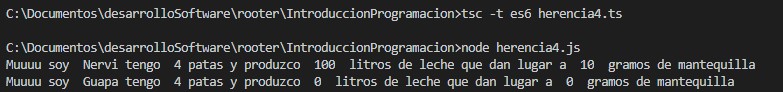


En el caso de la clase Vaca, añadimos un nuevo atributo privado litrosLeche, y sus correspondientes métodos get y set, pero, además, añadimos una función que calcula los gramos de mantequilla que produce una vaca en función de su producción de leche, que se estima en una décima parte. Por defecto, los litros de leche se inicializan a 0 en el constructor. Adicionalmente, modificamos la función saludo para que nos proporcione toda la información.

|  |
| --- |
| class Vaca extends Animal { private \_litrosLeche: number; constructor (name: string){ super(name); this.patas = 4; this.\_litrosLeche = 0;  } public get litrosLeche(): number { return this.\_litrosLeche;  }  public set litrosLeche(value: number) { this.\_litrosLeche= value;  }  public gramosMantequilla() : number { if (this.\_litrosLeche >0) { return this.\_litrosLeche/10;  } else return 0;  } saludo() { |

console.log ('Muuuu soy ', this. animalName, 'tengo ', this.patas , 'patas y produzco ', this.litrosLeche, ' litros de leche que dan lugar a ' , this.gramosMantequilla() , ' gramos de mantequilla'); } } let miVaca1 = new Vaca('Nervi'); miVaca1.litrosLeche = 100; miVaca1.saludo(); let miVaca2 = new Vaca('Guapa'); miVaca2.saludo();

La salida de programa sería:



Ejemplo 4: Atributos protegidos

Para completar el ejemplo, añadimos el campo protegido a la clase padre “sonido”. Este campo almacenará la información correspondiente al sonido de cada animal y podremos acceder a su información mediante la función pública getSonido.

Como podemos ver, el campo sonido es inicializado en los constructores de las clases Hijas como si fuera un atributo más, ya que está definido como un campo protegido, no privado, como en el caso de los otros dos atributos, en que se hacía necesario el uso de las funciones ***get*** y ***set***.

El ejemplo completo quedaría así:

abstract class Animal {

private \_animalName: string;

private \_patas: number;

protected sonido: string;

constructor (name: string){

this.\_animalName = name;

}

public get animalName(): string {

return this.\_animalName;

}

public set animalName(value: string) {

this.\_animalName= value;

}

public get patas(): number {

**5. Modificadores**

return this.\_patas;

}

public set patas(value: number) {

this.\_patas= value;

}

abstract saludo(): void;

public getsonido() {

console.log ('El sonido de ',this.\_animalName, 'es un ', this.son

ido);

}

}

class Perro extends Animal {

private \_cazador: boolean;

constructor (name: string){

super(name);

this.patas = 4;

this.\_cazador = false;

this.sonido = 'ladrido';

}

public get cazador(): boolean {

return this.\_cazador;

}

public set cazador(value: boolean) {

this.\_cazador= value;

}

saludo() {

console.log ('Guau soy ' , this. animalName, 'tengo ', this.patas

, 'patas y ', this.cazador ? 'Si' : 'No', ' soy cazador');

}

}

class Vaca extends Animal {

private \_litrosLeche: number;

constructor (name: string){

super(name);

this.patas = 4;

this.\_litrosLeche = 0;

this.sonido = 'mugido';

}

public get litrosLeche(): number {

return this.\_litrosLeche;

}

public set litrosLeche(value: number) {

this.\_litrosLeche= value;

}

public gramosMantequilla() : number {

if (this.\_litrosLeche >0) {

**5. Modificadores**

return this.\_litrosLeche/10;

} else

return 0;

}

saludo() {

console.log ('Muuuu soy ', this. animalName, 'tengo ', this.patas

, 'patas y produzco ', this.litrosLeche, ' litros de leche que dan lugar

a ' , this.gramosMantequilla() , ' gramos de mantequilla');

}

}

let miPerro1 = new Perro('Rober');

let miPerro2 = new Perro('Rhin');

let miVaca1 = new Vaca('Nervi');

let miVaca2 = new Vaca('Guapa');

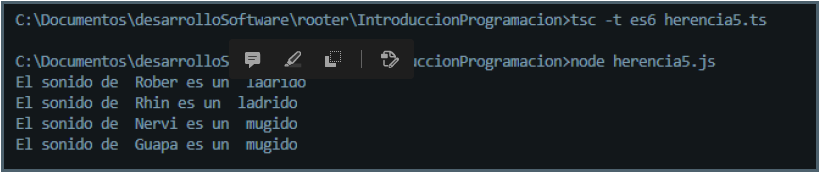
miPerro1.getsonido();

miPerro2.getsonido();

miVaca1.getsonido();

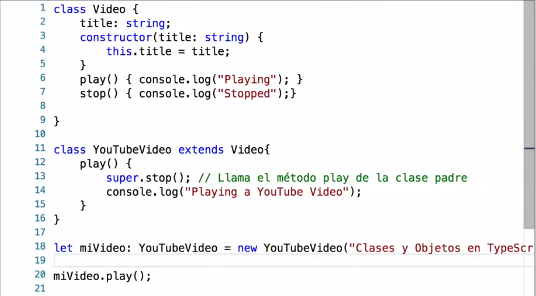
miVaca2.getsonido();

Y la salida de programa sería:



## Sobrecarga de métodos

En ocasiones querremos reemplazar las clases heredadas del padre.



Los constructores también se pueden sobrescribir, pero siempre que sobrescribamos un constructor tenemos que llamar al constructor del padre, con “super” (y con sus argumentos) y tratarlo como una función, no como un objeto.



## POLIMORFISMO

El concepto de polimorfismo se puede aplicar tanto a métodos como a atributos. Así, se pueden distinguir métodos polimórficos y atributos polimórficos.

* Los métodos polimórficos son aquellas funciones que pueden evaluarse o ser aplicadas a diferentes tipos de datos de forma indistinta.
* Los tipos polimórficos, por su parte, son aquellos tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

Veremos, en los siguientes apartados, las clases principales de polimorfismo más comúnmente reconocidas: polimorfismo de sobrecarga, polimorfismo paramétrico o dinámico, polimorfismo de subtipado y polimorfismo con interfaces.

### El polimorfismo de sobrecarga

Ocurre cuando las funciones del mismo nombre existen, pero tienen funcionalidad distinta (aunque similar).

Este polimorfismo se utiliza habitualmente en los métodos constructores, permitiendo definir un comportamiento u otro según los parámetros que se le pasen. *TypeScript* no permite la definición de funciones con el mismo nombre dentro de la misma clase, por lo que para realizar este polimorfismo de sobrecarga tendríamos que apoyarnos en un interfaz externo para simularlo; se verá más adelante en el apartado Polimorfismo con interfaces. El lenguaje Java, sin embargo, sí permite la sobrecarga en constructores de una clase, ya que permite que varios métodos se llamen igual dentro de la misma clase siempre que su lista de parámetros sea distinta, de la forma:

    constructor(int a){ … }

    contructor(int a, string b){ ….}

    constructor(int a, int b, int c){ … }

Aunque no forma parte del alcance de este curso, es importante conocerlo, ya que es habitual su uso. A continuación, se muestra un ejemplo de polimorfismo de sobrecarga en lenguaje Java.

|  |
| --- |
| //Demostración de Sobrecarga de constructores class MiClase{ int x;  MiClase(){  System.out.println("Dentro de MiClase()."); x=0;  }  MiClase(int i){  System.out.println("Dentro de MiClase(int)."); x=i;  }  MiClase(double d){  System.out.println("Dentro de MiClase(double)."); x=(int)d;  }  MiClase(int i, int j){  System.out.println("Dentro de MiClase(int, int).");  x=i\*j;  } }  class DemoSobrecargaConstructor{ public static void main(String[] args) {  MiClase t1=new MiClase();  MiClase t2=new MiClase(28);  MiClase t3=new MiClase(15.23);  MiClase t4=new MiClase(2,4);  System.out.println("t1.x: "+ t1.x);  System.out.println("t2.x: "+ t2.x);  System.out.println("t3.x: "+ t3.x);  System.out.println("t4.x: "+ t4.x); } } |

La salida del programa sería:

Dentro de MiClase().

Dentro de MiClase(int).

Dentro de MiClase(double). Dentro de MiClase(int, int).

t1.x: 0 t2.x: 28 t3.x: 15 t4.x: 8

### Polimorfismo paramétrico o dinámico

El polimorfismo paramétrico es la capacidad para definir varias funciones utilizando el mismo nombre, pero usando parámetros diferentes (distinto nombre y/o tipo). Como se vio anteriormente, *TypeScript* no permite la definición de funciones del mismo nombre bajo la misma clase, pero sí pueden existir en clases distintas. El método correcto a aplicar se selecciona automáticamente en función del tipo de datos pasados por parámetros.

El polimorfismo paramétrico es aquel en el que el código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja. Así, puede ser utilizado por todo tipo de datos que sea compatible.

Veamos un ejemplo: definir varios métodos con el mismo nombre adicion() que efectúa una suma de valores, valores que pueden ser números enteros o cadenas de caracteres….

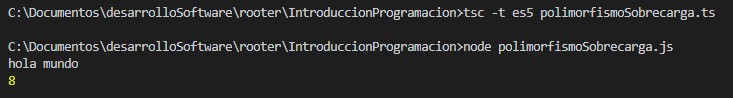
* El método int adicion (int,int) devolvería la suma de dos números enteros.
* El método char adicion (string, string) daría por resultado la concatenación de dos cadenas de caracteres.

Operador +: este ejemplo recuerda que el operador + (adición) se comporta de manera distinta según el tipo de datos. Cuando son números enteros, efectúa una suma y cuando se encuentra entre dos cadenas de caracteres, realiza una concatenación.

Continuando con este ejemplo, se puede crear una clase genérica de Operaciones, vacía de cualquier contenido, que simplemente se refiere a una operación suma, con hasta 3 parámetros de cualquier tipo. De esta clase heredan las clases Cadena y Numero que sobrecargan (implementan) la operación suma según los requerimientos del enunciado.

|  |
| --- |
| class Operaciones {          adicion(a?: any, b?: any, c?: number) { };      }      class Cadena extends Operaciones {          adicion(a: string, b: string) {              console.log(a + b);          }      }      class Numero extends Operaciones {          adicion(a: number, b: number, c: number) {              console.log(a + b + c);          }      let texto = new Cadena();      texto.adicion('hola ', 'mundo'); // hola mundo      let operacion = new Numero();      operacion.adicion(3, 2, 3); // 8 |

La salida por consola del programa anterior sería la siguiente:



### El polimorfismo de subtipado

También llamado polimorfismo de redefinición o polimorfismo de inclusión se refiere a un nombre que denota instancias de muchas clases diferentes relacionadas por alguna superclase (clase padre) común. Sería el caso de la clase Mamifero, por ejemplo.

El polimorfismo de subtipado se basa en la especialización a través de la herencia, de forma que permite trabajar con un objeto de una clase padre, aunque en realidad sea una instancia de diferentes clases hijas. Se llama especialización a redefinir un método (heredado de una clase padre) en las clases. Esto ocurrirá, por ejemplo, con los métodos mover: no es la misma forma de moverse la de un perro que la de un delfín, aunque ambos sean mamíferos.

Por lo tanto, se puede llamar a un método de un objeto sin tener que conocer su tipo intrínseco: este es el polimorfismo de subtipado. Así, al programar no es necesario conocer los detalles de todas las clases especializadas (hijas) de una familia de objetos, y se pueden enmascarar los detalles comunes en la clase padre (superclase).

Ahora vamos a continuar con el ejemplo de la clase Animal planteado en el capítulo de la herencia.

Se han declarado dos objetos de la clase Animal y se han asignado a dos instancias de objetos Perro y Vaca. Como no se pueden crear objetos de una clase abstracta, los objetos reales son instancias de las clases hijas. Al invocar el método saludo(), el método real al que se llama en cada caso es el que corresponde a cada instancia definida. Así, se puede comprobar en la salida del programa por consola, que en el caso de animal1 que se ha instanciado como Perro, el saludo es el que corresponde a la clase Perro y, en el caso de animal2, el saludo es el de la clase Vaca:

    abstract class Animal {

        private \_animalName: string; private \_patas: number;

constructor(name: string) {

            this.\_animalName = name;

        }

        public get animalName(): string {

            return this.\_animalName;

        }

        public set animalName(value: string) {

            this.\_animalName = value;

        }

public get patas(): number {

            return this.\_patas;

        }

public set patas(value: number) {

            this.\_patas = value;

        }

abstract saludo(): void;

    }

    class Perro extends Animal {

        constructor(name: string) {

            super(name);

            this.patas = 4;

        }

        saludo() {

            console.log('Guau soy ' + this.animalName);

        }

    }

    class Vaca extends Animal {

        constructor(name: string) {

            super(name);

            this.patas = 4;

        }

        saludo() {

            console.log('Muuuu soy ' + this.animalName);

        }

    }

    var animal1: Animal;

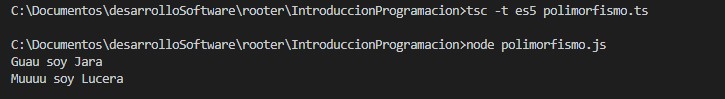
    var animal2: Animal;

    animal1 = new Perro('Jara');

    animal2 = new Vaca('Lucera');

    animal1.saludo();

    animal2.saludo();



Otro posible ejemplo, es el de programar un juego de ajedrez con los objetos rey, reina, alfil, caballo, torre y peón, cada uno de ellos heredando del objeto pieza. El método movimiento(), usando polimorfismo de subtipado, permitiría hacer el movimiento correspondiente a cada pieza, de acuerdo al objeto que lo llame, sin tener que verse conectado a ningún tipo de pieza en particular.

Como comentamos con anterioridad, *TypeScript* no permite definir varios métodos con el mismo nombre dentro de la misma clase. Por ello, en ocasiones se hace uso de la definición de interfaces que permiten resolver esta problemática.

Se plantea a continuación la definición de una clase Box, para la que se necesita disponer de dos constructores con parámetros diferentes.

Vamos a ver cómo lograrlo. Si intentamos programar el siguiente código:

El compilador nos indica que hay error porque, como se ha dicho, TypeScript no permite definir varios métodos con el mismo nombre dentro de la misma clase.



Para resolverlo, hay que definir una interface IBox con las propiedades de la clase que permitirá la manipulación en el constructor correspondiente, de la forma:

    interface IBox {

        x: number;

        y: number;

        height: number;

        width: number;

    }

    class Box {

        public x: number;

        public y: number;

        public height: number;

        public width: number;

        constructor(obj?: IBox) {

            this.x = obj && obj.x || 0

            this.y = obj && obj.y || 0

            this.height = obj && obj.height || 0

            this.width = obj && obj.width || 0;

        }

    }

    let caja = new Box();

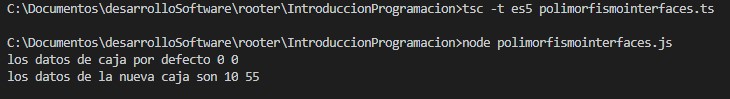
    console.log('los datos de caja por defecto', caja.x, caja.y);

    var micaja = <IBox>{ x: 10, y: 55 };

    let cajanueva = new Box(micaja);

    console.log('los datos de la nueva caja son',cajanueva.x,cajanueva.y);

Así, la salida de programa sería la siguiente, obteniendo dos cajas de dos tamaños distintos:



Los interfaces son muy utilizados en lenguaje TypeScript, por lo que vamos a ver otro ejemplo de uso, asociado a definir el comportamiento de ciertas clases. Gracias a los interfaces, podemos definir el comportamiento que deseamos que tenga una clase, obligando a implementar los métodos definidos en el interfaz a todas las clases que implementen dicho interfaz. En este caso, queremos definir varios motores distintos, de combustión y eléctricos, y ponerlos en marcha y pararlos.

    interface IMotor {

        start();

        stop();

    }

    class MotorCombustion implements IMotor {

        marca: string;

        modelo: string;

        año: number;

        tipocombustible: string;

        constructor(m: string, mod: string, a: number, c: string) {

            this.marca = m;

            this.modelo = mod;

            this.año = a;

            this.tipocombustible = c;

        };

        start() {

            console.log("arranca el motor", this.marca, ' ',

this.modelo, ' ', this.año);

        };

        stop() {

            console.log("para el motor", this.marca, ' ',

this.modelo, ' ', this.año);

        };

    }

    class MotorElectrico implements IMotor {

        marca: string;

        modelo: string;

        año: number;

        numerobaterias: number;

        constructor(m: string, mod: string, a: number, b: number) {

            this.marca = m;

            this.modelo = mod;

            this.año = a;

            this.numerobaterias = b;

        };

        start() {

            console.log("arranca el motor electrico", this.marca, ' ',

this.modelo, ' ', this.año);

        };

        stop() {

            console.log("para el motor electrico", this.marca, ' ',

this.modelo, ' ', this.año);

        };

    }

    let mimotor = new MotorCombustion('mercedes', 's30', 1995, 'diesel');

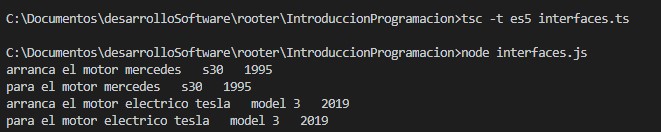
    mimotor.start();

    mimotor.stop();

    let tumotor = new MotorElectrico('tesla', 'model 3', 2019, 25);

    tumotor.start();

La salida de este programa en la consola será:



Se podría implementar este mismo ejemplo utilizando Herencia de la siguiente forma:

    abstract class IMotor {

        constructor() { };

        start() { };

        stop() { };

    }

    class MotorCombustion extends IMotor {

        marca: string;

        modelo: string;

        año: number;

        tipocombustible: string;

        constructor(m: string, mod: string, a: number, c: string) {

            super();

            this.marca = m;

            this.modelo = mod;

            this.año = a;

            this.tipocombustible = c;

        };

        start() {

            console.log("arranca el motor", this.marca, ' ',

this.modelo, ' ', this.año);

        };

        stop() {

            console.log("para el motor", this.marca, ' ',

this.modelo, ' ', this.año);

        };

    }

    class MotorElectrico extends IMotor {

        marca: string;

        modelo: string;

        año: number;

        numerobaterias: number;

        constructor(m: string, mod: string, a: number, b: number) {

            super();

            this.marca = m;

            this.modelo = mod;

            this.año = a;

            this.numerobaterias = b;

        };

        start() {

            console.log("arranca el motor electrico", this.marca, ' ',

this.modelo, ' ', this.año);

        };

        stop() {

            console.log("para el motor electrico", this.marca, ' ',

this.modelo, ' ', this.año);

        };

    }

    let mimotor = new MotorCombustion('mercedes', 's30', 1995, 'diesel');

    mimotor.start();

    mimotor.stop();

    let tumotor = new MotorElectrico('tesla', 'model 3', 2019, 25);

    tumotor.start();

Y, como se comprueba, la salida del programa sería la misma. Por tanto, no existe un consenso para identificar si una forma de implementación es mejor que la otra, aunque con Herencia se permiten hacer desarrollos más complejos que no se lograrían utilizando solamente las interfaces.

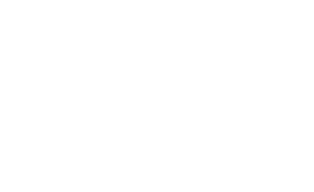
## Diferencia entre polimorfismo y sobrecarga

Aunque este curso está orientado al uso de lenguaje *TypeScript* y el uso del polimorfismo está limitado, se ha querido incluir un primer contacto con este concepto. En programación, es muy habitual que se hable de polimorfismo de sobrecarga e incluso se hable de las diferencias entre sobrecarga y polimorfismo por estar dichos conceptos muy relacionados.

Como se ha visto, un método está sobrecargado si dentro de una única clase existen varias declaraciones de dicho método con el mismo nombre, pero con parámetros distintos (de distinto tipo, o distinto número de parámetros). Como ya se ha visto anteriormente, no todos los lenguajes de programación permiten realizar sobrecarga de métodos.

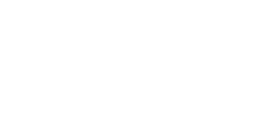
En cambio, el polimorfismo consiste en redefinir un método de una clase padre en una clase hija, como se ha visto en esta nota técnica. Es decir, se utilizan métodos de distintas clases que cumplen la misma función en cada una.

Otra diferencia importante es que la sobrecarga se resuelve en tiempo de compilación utilizando los nombres de los métodos y los tipos de sus parámetros. En cambio, el polimorfismo se resuelve en tiempo de ejecución del programa; esto es, mientras se ejecuta, en función de la clase a la que pertenece el objeto.

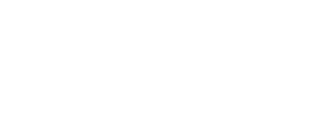


HERENCIA

SIMPLE



Isósceles



POLIMORFISMO

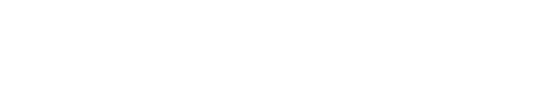
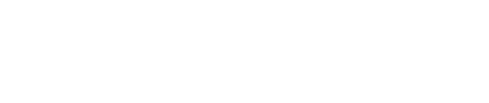
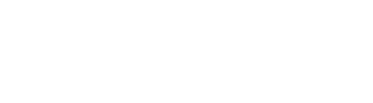


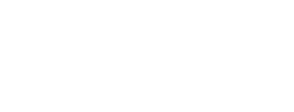
Figura Plana



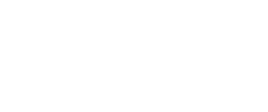
Rectángulo



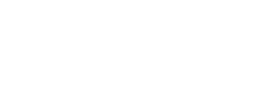
Triángulo



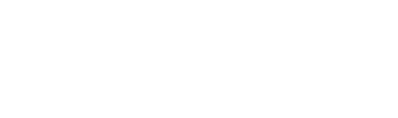
Cuadrado



Equilátero



Escaleno



CLASE ABSTRACTA

STATIC

Los métodos

# MODULARIDAD

La modularidad, en programación, consiste en dividir un determinado código en módulos. El sistema original se divide en un conjunto de piezas que, aunque independientes, pueden relacionarse entre sí.

En la programación orientada a objetos cada clase se guarda dentro de un módulo, y también las interfaces, como se verá en la siguiente nota técnica.

La modularidad, ayudada por la abstracción que ya se vio en otra nota técnica, permite:

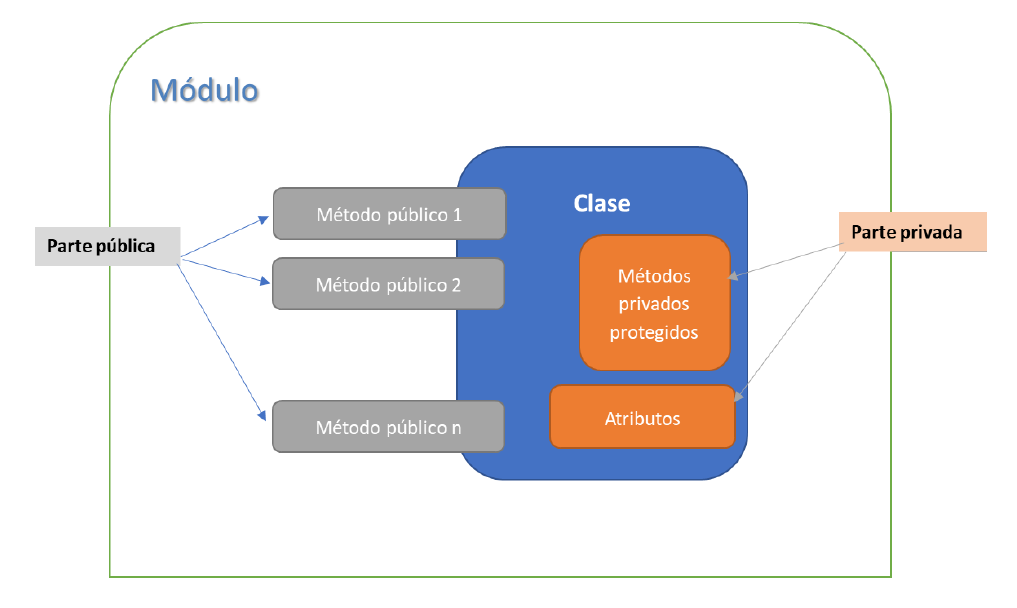
• El ocultamiento de los detalles en programación, facilitando la comprensión global del código.

• Los cambios en la implementación de un módulo no afectan al resto del sistema, por tanto, facilita el mantenimiento del código.

• Que los módulos puedan reutilizarse en distintas aplicaciones.

• Que cada clase/módulo pueda ser probado de forma casi independiente de las demás clases, facilitando las pruebas unitarias.

Las clases tienen una parte visible (pública) y una parte oculta (privada o protegida) que no será accesible desde fuera de la clase y, por tanto, no será accesible desde fuera del módulo.



**2. Aspect**

**Aspectos importantes**

A la hora de realizar una correcta división en módulos, es importante tener en cuenta una serie de consideraciones que facilitarán la división. A continuación, se enumeran los aspectos más importantes a tener en cuenta.

Lo habitual es que una clase:

• Tenga todos sus atributos privados.

• Solo tenga los métodos públicos estrictamente necesarios.

De esta forma, el estado del objeto solo podrá cambiar a través de unos puntos de entrada bien definidos: sus métodos públicos.

**Capacidad de descomponer un sistema complejo**

Se trata de descomponer un programa en subprogramas, dividiendo el problema general en problemas más pequeños y que se resuelvan más fácilmente.

Esta capacidad se basa en uno de los principios fundamentales de la programación: el principio de “divide y vencerás”, el cual supone que si se divide un problema en pequeñas secciones (algoritmos) éste puede ser resuelto de forma más eficiente que si se intenta resolver entero.

**Capacidad de componer a través de módulos**

Se trata de componer el programa desde los problemas más pequeños (las divisiones), completando y resolviendo el problema general gracias a la utilización de módulos más pequeños.

Este aspecto es muy importante cuando se crea software reutilizando código, a partir de módulos ya existentes. Por ello, cada vez que se crea un módulo, hay que pensar que pueda ser reusable.

**Comprensión del sistema en partes**

Disponer de las partes separadas de un sistema ayuda a comprender mejor su código y también ayuda a modificarlo. Trabajar con un único módulo puede resultar un caos.

**3. Export**

**/ Import**

## IMPORT / EXPORT

La definición de una clase y sus atributos y métodos se ha estudiado en las notas técnicas previas. En esta nota técnica dedicada a la modularidad, se van a preparar esas clases para que puedan ser usadas por otros módulos.

En *TypeScript* se utilizan los comandos **export** e **import** como se explica a continuación.

**Export** permite indicarle a una clase que puede ser usada por otro módulo. Desde el módulo que se use, solo podrá acceder a los atributos y métodos que hayan sido definidos como públicos.

Con export lo que hacemos es exportar una clase en concreto, no el archivo .ts, es decir, con export, exponemos la clase para que esté accesible desde cualquier otro módulo de nuestro proyecto. Esa es la principal diferencia con los namespaces. Con los namespaces Typescript carga todo el archivo.

Tengo que poner export en todo aquel elemento que quiero exportar, esa una clase, una función, variable…etc:

    export class miClase{

        //Código de la clase

    }

    export function miFunction(nombre:string){

        console.log('Hola ${nombre}')

    }

    export var edad = 42;

    export var activo = 0;

Para poder usar un módulo desde otro módulo recurriremos al comando **import** seguido del nombre de la clase y la ruta donde se encuentra. Si está en el mismo directorio pondremos ./.

Si quiero importar varios elementos tengo que hacer un import por elemento.

import { miClase} from './miArchivo';

Podemos llamar a varios elementos en un solo import:

import { miClase, miFuncion, miVariable} from './miArchivo';

Es habitual que las clases estén organizadas en una estructura de directorios, con lo que el nombre de la clase puede contener la ruta completa de carpetas donde se encuentra el archivo de implementación.

Si nuestro archivo se llama index.ts podemos poner simplemente el directorio donde se encuentra el archivo, sin poner el nombre del archivo. Esto funciona sólo si el archivo se llama index, de lo contrario hay que especificar el nombre del archivo:

import { miClase, miFuncion, miVariable} from './miDirectorio/';

También podemos importar todo lo “importable” de un archivo y hacer uso de los alias:

import \* as miAlias from './miArchivo';

Después en el archivo donde lo vamos a usar tenesitaremos aplicar la nomenclatura del punto:

miAlias.miFuncion();

También podemos hacer una exportación/importación por defecto:

    export function miFuncion(){

        //Cuerpo de la función

    }

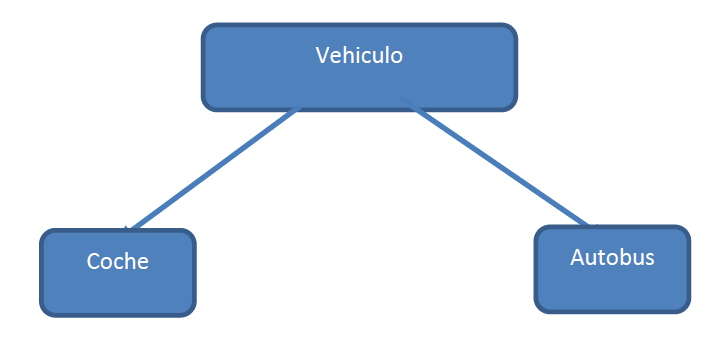
    export default miFuncion;

Luego en el archivo receptor lo importo poniendo un alias:

import miAlias from './miArchivo';

Y ya lo puedo usar poniendo la nomenclatura del punto.

En el siguiente ejemplo trabajaremos con el siguiente diagrama de clases:



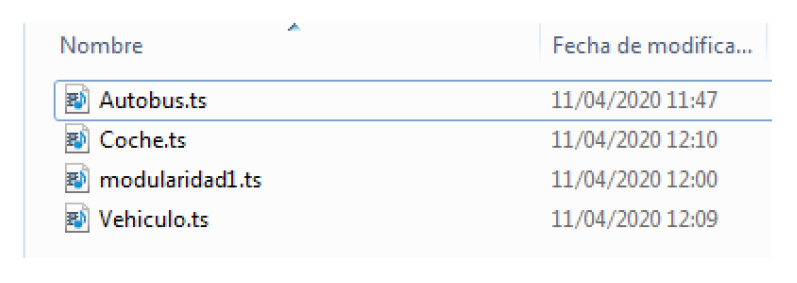
La clase Vehiculo es la Superclase de la que heredan las clases Coche y Autobús. A diferencia de las notas técnicas anteriores, en las que teníamos todo el código en el mismo fichero, ahora cada clase estará alojada en un fichero TypeScript (.ts) diferente. Si necesitamos, por ejemplo, una nueva clase “Furgoneta”, deberíamos crear un nuevo fichero .ts sin necesidad de tocar el código de ninguno de los ficheros que ya tenemos definidos.

Todas las clases serán definidas como **export**.

• La clase Vehículo necesita ser exportable para que pueda ser utilizada desde las clases Coche y Autobus.

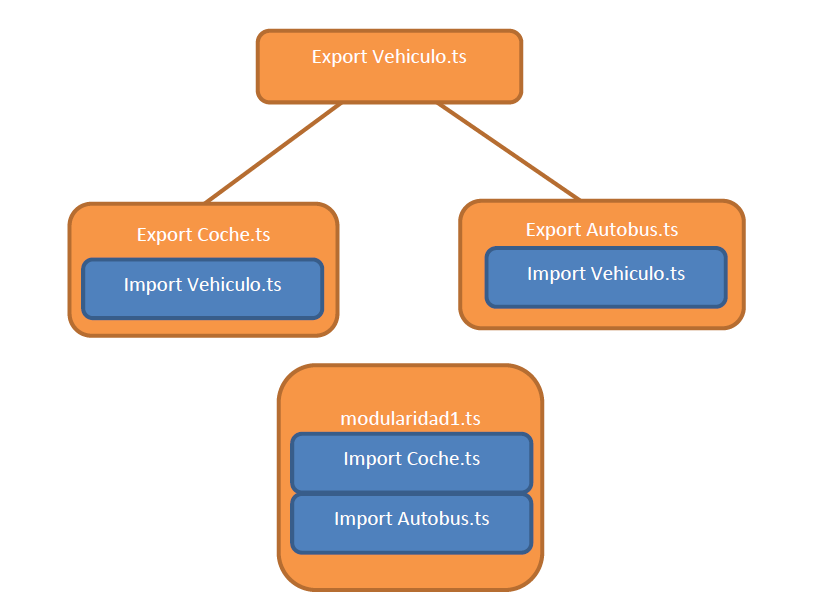
• Las clases Coche y Autobus necesitan ser exportables para poder ser usadas (instanciadas) desde el programa principal desde donde se manejan los objetos. En nuestro caso el fichero se ha denominado “modularidad1.ts”.

El esquema de ficheros que tenemos en el siguiente:



Como podemos ver, están todos agrupados en el mismo directorio, pero se podría trabajar teniéndolos en rutas diferentes.

El contenido de los diferentes ficheros queda como sigue:



Export Coche.ts

Export Autobus.ts

**Clase Vehiculo**

La clase Vehiculo se ha modificado para declarar los atributos de la clase como campos privados. Además, solo se emplean métodos ‘set’ para aquellos atributos que pueden ser modificados. No es el caso de la matrícula del vehículo, por ejemplo, ya que la matrícula de un vehículo no debería modificarse.

Además, se añade la función cargaUtil que calcula la carga útil de un vehículo en función de su tara y su pma.

Como vemos en el código de la clase, lo primero que aparece es la palabra reservada Export, para hacer que la clase Vehiculo pueda ser utilizada por sus subclases:

    export abstract class Vehiculo {

        private \_marca: string;

        private \_modelo: string;

        private \_matricula: string;

        private \_pma: number;

        private \_tara: number;

        constructor(a: string,b: string,c: string,d: number, e: number) {

            this.\_marca = a;

            this.\_modelo = b;

            this.\_matricula = c;

            this.\_pma = d;

            this.\_tara = e;

        }

        public get marca(): string {

            return this.\_marca;

        }

        public set marca(value: string) {

            this.\_marca = value;

        }

        public get modelo(): string {

            return this.\_modelo;

        }

        public set modelo(value: string) {

            this.\_modelo = value;

        }

        public get matricula(): string {

            return this.\_matricula;

        }

        public get pma(): number {

            return this.\_pma;

            Export / Import

        }

        public set pma(value: number) {

            this.\_pma = value;

        }

        public get tara(): number {

            return this.\_tara;

        }

        public set tara(value: number) {

            this.\_tara = value;

        }

        public encender(): void {

            console.log("Encendido vehiculo");

        }

        public apagar(): void {

            console.log("Apagado vehiculo");

        }

        abstract acelerar(): void;

        abstract frenar(): void;

        public cargaUtil(): number {

            let value: number;

            value = this.\_pma - this.\_tara;

            if (value > 0) {

                return value;

            } else

                return 0;

        }

    }

**Clase Coche**

En este caso, se modifica el constructor de la clase para que llame al constructor de la clase padre. Solamente se implementan las funciones abstractas que definen la clase padre.

En la primera línea de código se importa la clase Vehiculo:

    import { Vehiculo } from './Vehiculo';

    export class Coche extends Vehiculo {

        constructor(a: string,

b: string,

c: string,

d: number,

e: number) {

            super(a, b, c, d, e);

        }

        public acelerar(): void {

            console.log("Acelerar coche");

        }

        public frenar(): void {

            console.log("Frenar coche");

        }

    }

**Clase Autobus**

En la definición de la clase Autobus, hay que importar la clase Vehiculo en la primera línea para que se pueda utilizar. Las funciones provenientes de la clase padre se implementan como abstractas y las demás, las nuevas funcionalidades, en la clase hija.

Esto se ve, por ejemplo, en las funciones que imprimen en pantalla: las funciones encender y apagar de la clase Autobus imprimen un mensaje diferente del que escriben las mismas funciones en la clase Vehiculo. A la hora de llamar a dichas funciones, con una instancia de la clase Coche o de la clase Autobus, se podrán comprobar las diferencias.

    import { Vehiculo } from './Vehiculo';

    export class Autobus extends Vehiculo {

        constructor(a: string, b: string, c: string, d: number, e: number) {

            super(a, b, c, d, e);

        }

        public encender(): void {

            console.log("Encendido autobús");

        }

        public apagar(): void {

            console.log("Apagado autobús");

        }

        public acelerar(): void {

            console.log("Acelerar autobús");

        }

        public frenar(): void {

            console.log("Frenar autobús");

        }

    }

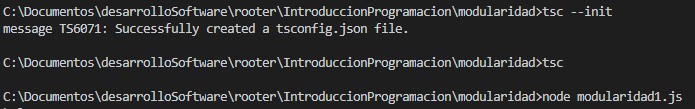
# COMPILACIÓN

Cuando se utilizan módulos, el compilador debe conocer el orden de compilación y las directrices a la hora de generar el código. En el caso de TypeScript, todos los ficheros generan un fichero del tipo JavaScript (.js).

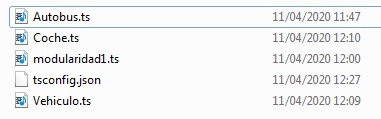
Lo más sencillo es darle estas directrices al compilador a través de un fichero denominado tsconfig.json. Este fichero incluye todas las opciones necesarias para compilación y ejecución de un proyecto.

Este fichero se puede generar de forma manual, pero lo más sencillo es que lo genere el propio compilador.

Estando en la ruta donde tenemos todos los ficheros que hemos generado para el proyecto ejecutamos la orden:

tsc -–init

De esta forma tan sencilla, se ha creado un tsconfig.json con toda la información necesaria para que se compilen todos los módulos y podamos ejecutar luego el código que incluyamos en el programa principal (modularidad1.ts) con la ayuda del comando “node”.



La compilación y ejecución del código quedaría como sigue:

**Programa principal**

El contenido del programa principal que se muestra a continuación sirve para mostrar el uso de algunas funciones y atributos de las clases comentadas anteriormente.

Así, el siguiente código se encuentra dentro del fichero que hemos denominado modularidad1.ts.

import { Coche } from "./Coche"; import { Autobus } from "./Autobus";

console.log('------- Iniciamos ejecución del programa ------- ');

let miCoche1 = new Coche('Tesla', 'model 3', 'LE-1010-FG', 1000, 950);

console.log('La carga util de ', miCoche1.marca, '', miCoche1.modelo, ' es : ', miCoche1.cargaUtil());

miCoche1.pma = 3000;

console.log('La carga util de ', miCoche1.marca, '', miCoche1.modelo, ' es : ', miCoche1.cargaUtil());

miCoche1.encender();

miCoche1.acelerar();

miCoche1.frenar();

miCoche1.apagar();

let miAutobus1 = new Autobus('IVECO', 'MX230', '4555FGK', 10000, 9000);

console.log('La carga util de ', miAutobus1.marca, '', miAutobus1.modelo, ' es : ', miAutobus1.cargaUtil());

miAutobus1.tara = 8000;

console.log('La carga util de ', miAutobus1.marca, '', miAutobus1.modelo, ' es : ', miAutobus1.cargaUtil());

miAutobus1.encender();

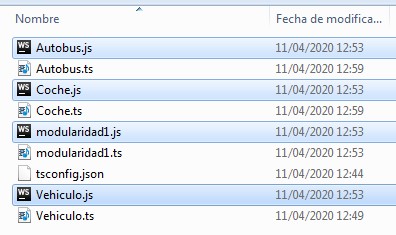
miAutobus1.acelerar();

miAutobus1.frenar();

miAutobus1.apagar();

console.log('------  Finalizamos ejecución del programa ------');

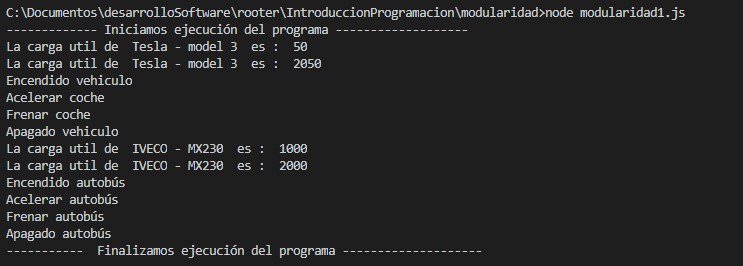
Al compilar con el comando tsc, se generan los ficheros .js correspondientes.



Posteriormente ejecutando nuestro programa con el comando:

node modularidad1.js

La salida obtenida en pantalla sería la siguiente:



En el desarrollo de aplicaciones informáticas, es habitual tener necesidades de programación que ya han sido resueltas por otros programadores y que suelen estar disponibles para su uso. Las licencias varían, pudiendo ser OpenSource, de pago, limitado para uso comercial, etc. El tema de las licencias no es un tema que se vaya a abordar en este curso, pero lo que sí debemos mencionar es cómo utilizar módulos y librerías que no hemos desarrollado nosotros y que pueden sernos útiles a la hora de dar solución a nuestros problemas.

A continuación, veremos un ejemplo de cómo utilizar la librería de funciones matemáticas para resolver problemas dentro de nuestras clases. En la actualidad, existen infinidad de soluciones publicadas en internet que pueden dar solución a muchos de los problemas a los que se enfrentan los desarrolladores en el día a día.

Lo primero que haremos será localizar la librería que necesitamos y ver qué debemos hacer para usarla. En el caso que nos ocupa localizamos la librería math en el siguiente enlace:

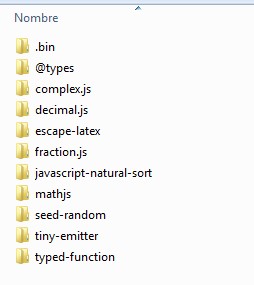
<https://mathjs.org/docs/getting_started.html>

Como podemos ver, nos indica que tendremos que instalarla utilizando los comandos:

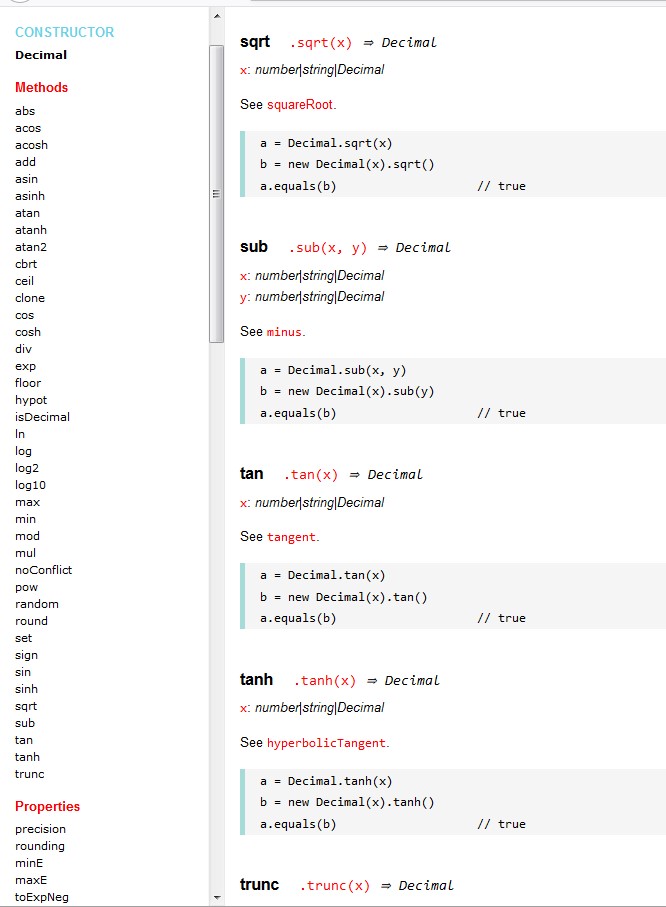
npm install mathjs

npm install @types/mathjs

Estos dos comandos han creado dentro de nuestra carpeta del proyecto una carpeta node\_modules donde se ha copiado el código de esta librería:



Para revisar la documentación podríamos acceder, por ejemplo, al fichero node\_modules\decimal.js\doc\API.html y podríamos ver que existen infinidad de funciones con las que podríamos trabajar con valores decimales y que están documentadas en este fichero.



De esta forma, ya se podrían utilizar las funciones que se incluyen en la librería matemática. Para ello, basta importar la función que queramos en el código del programa principal. Por ejemplo, la función raíz cuadrada:

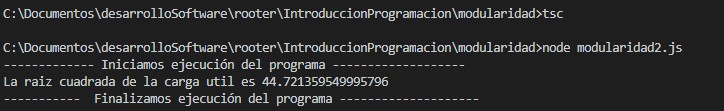
import {sqrt} from "mathjs"

Y la llamamos desde el programa principal:

console.log('La raiz cuadrada de la carga util es', sqrt(miAutobus1.carga

Util()));

Tras compilar con tsc y ejecutar con node, tendríamos:



Y la salida del programa en la pantalla quedaría:



# CONFIGURACIÓN

### Contexto de compilación

El TypeScript Compilation Context es un concepto que se usa para agrupar los archivos de TypeScript que diseccionarán y analizarán nuestro código para verificar que es válido o los posibles errores que contenga. El contexto de compilación, además de los archivos, contiene la información acerca de que opciones de compilación se están usando.

El archivo de configuración donde se almacenan las opciones de compilación para poder compilar el proyecto actual es el fichero “tsconfig.json”. Al crear un nuevo proyecto en TypeScript, nuestro primer paso debe ser añadir el archivo “tsconfig.json”, este archivo se almacena en el mismo directorio donde guardemos nuestro código, directorio root de nuestro proyecto. Este archivo es una de las mejores maneras para definir nuestro contexto de compilación. La sintaxis de este archivo sigue el formato JSON.

Podremos compilar nuestro proyecto TypeScript, utilizando el compilador “tsc”, de dos maneras diferentes:

- Invocando el comando tsc sin archivos de input. En este caso todas las opciones de compilación estarían almacenadas en el archivo tsconfig.json, que el compilador buscaría en el directorio actual.

> tsc

- Invocando el comando tsc sin archivos de input y con la opción --project (abreviado –p) e indicando a continuación el path del proyecto que se va a compilar y en el que estará el archivo tsconfig.json. O el path completo a un archivo de extensión json con la configuración. Si se especifica un archivo de configuración, se ignora cualquier posible archivo tsconfig.json

>tsc –p ./ruta\_al\_proyecto

### Creación tsconfig.json

Podremos crear un archivo según las indicaciones anteriores con un editor de texto, y lo guardaremos en la ruta del proyecto con el nombre tsconfig.json. Si el archivo es vacío, tomará los parámetros por defecto en la compilación. Un archivo vacío sería {}

Otra opción es directamente por la línea de comandos, usando la siguiente sentencia:

>tsc –init message

TS6071: Successfully created a tsconfig.json file.

Lo que nos creará un archivo de configuración por defecto, tsconfig.json, con la siguiente apariencia.

|  |
| --- |
| {  "compilerOptions": {  /\* Basic Options \*/  // "incremental": true,  /\* Enable incremental compilation \*/ "target": "es5",  /\* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019' or 'ESNEXT'. \*/ "module": "commonjs",  /\* Specify module code generation: 'none', 'commonjs', 'amd', 'system',  'umd', 'es2015', or 'ESNext'. \*/ // "lib": [],  /\* Specify library files to be included in the compilation. \*/ // "allowJs": true,  /\* Allow javascript files to be compiled. \*/ // "checkJs": true,  /\* Report errors in .js files. \*/ // "jsx": "preserve",  /\* Specify JSX code generation: 'preserve', 'react-native', or 'react'.  \*/  // "declaration": true,  /\* Generates corresponding '.d.ts' file. \*/ // "declarationMap": true,  /\* Generates a sourcemap for each corresponding '.d.ts' file. \*/ // "sourceMap": true,  /\* Generates corresponding '.map' file. \*/ // "outFile": "./",  /\* Concatenate and emit output to single file. \*/ // "outDir": "./",  /\* Redirect output structure to the directory. \*/ // "rootDir": "./",  /\* Specify the root directory of input files. Use to control the output directory structure with --outDir. \*/ // "composite": true,  /\* Enable project compilation \*/ // "tsBuildInfoFile": "./",  /\* Specify file to store incremental compilation information \*/ // "removeComments": true,  /\* Do not emit comments to output. \*/ // "noEmit": true,  /\* Do not emit outputs. \*/  // "importHelpers": true,  /\* Import emit helpers from 'tslib'. \*/ |

|  |
| --- |
| // "downlevelIteration": true,  /\* Provide full support for iterables in 'for-of', spread, and destructuring when targeting 'ES5' or 'ES3'. \*/ // "isolatedModules": true,  /\* Transpile each file as a separate module (similar to  'ts.transpileModule'). \*/  /\* Strict Type-Checking Options \*/  "strict": true,  /\* Enable all strict type-checking options. \*/ // "noImplicitAny": true,  /\* Raise error on expressions and declarations with an implied 'any' type. \*/  // "strictNullChecks": true,  /\* Enable strict null checks. \*/  // "strictFunctionTypes": true,  /\* Enable strict checking of function types. \*/ // "strictBindCallApply": true,  /\* Enable strict 'bind', 'call', and 'apply' methods on functions. \*/ // "strictPropertyInitialization": true,  /\* Enable strict checking of property initialization in classes. \*/ // "noImplicitThis": true,  /\* Raise error on 'this' expressions with an implied 'any' type. \*/ // "alwaysStrict": true,  /\* Parse in strict mode and emit "use strict" for each source file. \*/  /\* Additional Checks \*/  // "noUnusedLocals": true,  /\* Report errors on unused locals. \*/ // "noUnusedParameters": true,  /\* Report errors on unused parameters. \*/ // "noImplicitReturns": true,  /\* Report error when not all code paths in function return a value. \*/ // "noFallthroughCasesInSwitch": true,  /\* Report errors for fallthrough cases in switch statement. \*/  /\* Module Resolution Options \*/  // "moduleResolution": "node",  /\* Specify module resolution strategy: 'node' (Node.js) or 'classic' (TypeScript pre-1.6). \*/ // "baseUrl": "./",  /\* Base directory to resolve non-absolute module names. \*/ // "paths": {},  /\* A series of entries which re-map imports to lookup locations relative to the 'baseUrl'. \*/  // "rootDirs": [], |

|  |
| --- |
| /\* List of root folders whose combined content represents the structure of the project at runtime. \*/ // "typeRoots": [],  /\* List of folders to include type definitions from. \*/ // "types": [],  /\* Type declaration files to be included in compilation. \*/ // "allowSyntheticDefaultImports": true,  /\* Allow default imports from modules with no default export. This does not affect code emit, just typechecking. \*/ "esModuleInterop": true  /\* Enables emit interoperability between CommonJS and ES Modules via creation of namespace objects for all imports. Implies  'allowSyntheticDefaultImports'. \*/ // "preserveSymlinks": true,  /\* Do not resolve the real path of symlinks. \*/ // "allowUmdGlobalAccess": true,  /\* Allow accessing UMD globals from modules. \*/  /\* Source Map Options \*/  // "sourceRoot": "",  /\* Specify the location where debugger should locate TypeScript files instead of source locations. \*/ // "mapRoot": "",  /\* Specify the location where debugger should locate map files instead of generated locations. \*/  // "inlineSourceMap": true,  /\* Emit a single file with source maps instead of having a separate file.  \*/  // "inlineSources": true,  /\* Emit the source alongside the sourcemaps within a single file; requires '--inlineSourceMap' or '--sourceMap' to be set. \*/  /\* Experimental Options \*/  // "experimentalDecorators": true,  /\* Enables experimental support for ES7 decorators. \*/ // "emitDecoratorMetadata": true,  /\* Enables experimental support for emitting type metadata for decorators. \*/ }  } |

En el fichero creado solo se crea la propiedad compilerOptions con sus opciones por defecto y todas las posibilidades que permite. En los siguientes apartados se muestran las propiedades más habituales, que pueden ser interesantes.

### Propiedades

#### Includes / excludes

El uso de las propiedades includes y excludes nos permite añadir o excluir archivos TypeScript del proceso de compilación. Estas propiedades se añadirían como otro elemento adicional al fichero tsconfig.json

|  |
| --- |
| {  "compilerOptions": {  "target": "es5",  "module": "commonjs",  "strict": true,  "esModuleInterop": true  },  "include": [  "src/\*\*/\*" ],  "exclude": [  "node\_modules",  "\*\*/\*.spec.ts"  ]  } |

Disponemos de una serie de expresiones regulares, de cara a indicar los paths de los includes/excludes:

* ‘\*’ sustituye a cero o varios caracteres (excluyendo separadores de directorio/).
* ? sustituye un caracter (excluyendo el separador de directorio/).
* \*\*/ sustituye recursivamente cualquier subdirectorio.

La opción por defecto de include es incluir todos los archivos TypeScript (.ts, .d.ts y .tsx). Si en la ruta indicamos como nombre de archivo \* o .\* indicamos que el compilador actuará únicamente con los archivos con extensiones soportadas TypeScript y JavaScript (si está habilitado).

#### compilerOptions

Con el uso de esta propiedad (opcional) podemos especificar distintas opciones a la hora de compilar nuestro proyecto, podríamos indicar estas opciones por línea de comandos o incluirlas en el archivo tsconfig.json. Por ejemplo, utilizando el siguiente fichero json con los archivos main.ts y ficha.ts utilizados en la nota técnica anterior.

|  |
| --- |
| {  "compilerOptions": {  "target": "es5",  "module": "system",  "strict": true,  "esModuleInterop": true,  "noImplicitAny": true,  "removeComments": true,  "preserveConstEnums": true,  "outDir": "build",  "sourceMap": true  },  "include": [  "src/\*\*/\*"  ],  "exclude": [  "node\_modules",  "\*\*/\*.spec.ts"  ]  } |

En src dejamos los fuentes TypeScript y en build quedarán los ficheros compilados .js y js.map:



Al ejecutar “tsc” se tendría en build:



Es una propiedad opcional que, en caso de omisión, se tomarán para la compilación los parámetros indicados en el archivo tsconfig por defecto.

Veamos un resumen con las principales opciones disponibles para esta propiedad:

--allowJs: indica si se permite la compilación de archivos JavaScript.

--alwaysStrict: parsea (analiza) en modo estricto.

--baseUrl: indica el directorio base.

--build—b: Construye el proyecto y todas sus dependencias. Nos permite hacer construcciones incrementales inteligentes. Construir es compilar, enlazar y empaquetar nuestro código a una forma ejecutable.

--declaration—d: Genera un archive de declaración con extensión .d.ts.

--diagnostics: Muestra la información de diagnóstico.

--experimentalDecorators: habilita el uso de decoradores experimentales para los ES Decorators.

--help –h: muestra la ayuda.

--init: Inicializa el proyecto TypeScript y crea un tsconfig.json.

--isolatedModules: Se usa para transpilar cada archive en un módulo separado.

--lib: indica el listado de librerías a incluir en la compilación.

--module—m: indica el tipo de módulo de salida, por ejemplo: "CommonJS" (valor por defecto), "System", "ES6", "ES2015" or "ESNext.".

--moduleResolution: determina como se resuelven los módulos. Por ejemplo: Node para node.js.

--noEmitOnError: no emite errores, aunque se detecten en la compilación.

--outDir: redirige la estructura resultante de la compilación al path indicado.

--project –p: compila el proyecto que indiquemos con: bien con el archivo de configuración json, o el path al directorio que contiene tsconfig.json.

--showConfig: muestra la configuración usada en la compilación.

--sourceMap: genera el archivo .map correspondiente. Es útil para la depuración de errores.

--target--t: Especifica la versión de ECMAScript target por defecto "ES3".

--watch—w: Lanza el compilador en modo vigilancia (watch), de modo que cuando cambia cualquier de los archivos fuente, se lanza el proceso de compilación y se generan los archivos JS transpilados de nuevo. No debería usarse con otras opciones salvo con --verbose.

#### compileOnSave

Esta propiedad se usa en los Entornos de Desarrollo Integrados (IDE) para compilar los archivos TypeScript y generar de forma automática la salida. Esta propiedad le indica al IDE que debe compilar el proyecto cada vez que se guarde, siguiendo la configuración establecida en el fichero de configuración.

### Configuración heredada

Un archivo tsconfig.json puede heredar alguna configuración de otro archivo, usando la propiedad extends seguida del path del archivo del que se va a heredar. Si se produjera un bucle de herencias entre archivos, se lanzaría un mensaje de error.

La configuración del archivo nuevo se carga en primer lugar, a continuación, se cargaría la configuración heredada sobrescribiendo la original, si se diera el caso. Aquellas referencias relativas se cargarían respecto a la ruta del archivo que la contiene originalmente.

Ejemplo:

configs/padre.json:

{

"compilerOptions": {

"noImplicitAny": true,

"strictNullChecks": true

}

}

tsconfig.json:

{

"extends": "./configs/padre",

"files": [

"main.ts",

"supplemental.ts"

]

}

tsconfig.ampliado.json:

{

"extends": "./tsconfig",

"compilerOptions": {

"strictNullChecks": false

}

}

### Referencias

El uso de referencias (introducido en la versión 3.0 de TypeScript) permite una mejora en el tiempo de construcción de nuestro proyecto, realiza una separación lógica entre componentes y nos permite organizar nuestro código de una manera mejor, pudiéndolo separar en proyectos.

La propiedad references dentro del archivo tsconfig.json, nos delimita un array de objetos que nos pasan proyectos por referencia. Dentro de cada entrada del array indicaremos el path al archivo de configuración del proyecto que queremos pasar. Al importar un proyecto, importamos también los módulos referenciados por el mismo desde el archivo de salida (.d.ts), si el proyecto también genera otro archivo de salida (.d.ts), este también se importará. Además, si no se ha construido el proyecto referenciado previamente, se hará de forma automática

Para poder usar referencias, se han de cumplir unos requisitos:

* Debemos tener habilitada la propiedad composite, de modo que TS pueda encontrar fácilmente los archivos de salida.
* Podemos tener habilitada la propiedad declarationMap, que con los editores compatibles nos permitirá navegar por nuestro código de forma transparente.
* Podemos anteponer la salida del proyecto pasado como referencia usando la propiedad prepend. Esto no situaría la salida del proyecto pasado por referencia sobre la salida del proyecto actual.

{

"compilerOptions": {

// Opciones de compilación

},

"references": [

{ "path": "../ruta\_proyecto\_referencia(tsconfig.json)" }

]

}

## Archivo MAP

Los archivos “.map” son archivos de mapeo que permiten a las herramientas hacer el mapeo entre el código JavaScript y el código fuente TypeScript. Son útiles para depurar errores, ya que varias herramientas como VisualStudio o herramientas de Chrome, nos permiten usar estos archivos en lugar del JS generado por el compilador.

## OTRAS OPCIONES DE COMPILACIÓN

Además de la compilación manual, que acabamos de ver, existen otras opciones de compilación: herramientas de construcción (building) que nos permiten automatizar la transformación y la generación de un único archivo con nuestro código.

La llamada a estas herramientas se suele hacer por la línea de comandos, ya sea desde nuestro entorno de desarrollo (IDE) o desde fuera del mismo. Suelen disponer de herramientas de scripting para automatizar todo el proceso de construir nuestro proyecto (compilación, dependencias, empaquetar código binario, pruebas, etc.)

Estas herramientas son especialmente útiles en proyectos grandes, ya que es tedioso controlar qué partes de nuestro proyecto tenemos que construir en cada momento.

Existen varias herramientas que se integran con TypeScript, enumeremos las principales:

* Gulp
* Bower
* Duo
* Browserify
* Babel
* Grunt
* Jspm