

**TYPESCRIPT**

**ADOLFO SANZ DE DIEGO**

**PRONOIDE**

# **2 ACERCA DE**

## 2.1 AUTOR

- Adolfo Sanz De Diego
  - Blog: [asanzdiego.blogspot.com.es](http://asanzdiego.blogspot.com.es)
  - Correo: [asanzdiego@gmail.com](mailto:asanzdiego@gmail.com)
  - GitHub: [github.com/asanzdiego](https://github.com/asanzdiego)
  - Twitter: [twitter.com/asanzdiego](https://twitter.com/asanzdiego)
  - LinkedIn: [in/asanzdiego](https://in/asanzdiego)
  - SlideShare: [slideshare.net/asanzdiego](https://slideshare.net/asanzdiego)

## 2.2 LICENCIA

- Esta obra está bajo una licencia:
  - Creative Commons Reconocimiento-CompartirIgual 3.0

# **3 INTRODUCCIÓN**

## 3.1 INTRODUCCIÓN

- Lenguaje de programación de alto nivel, de código abierto, desarrollado por Microsoft.
- Cuenta con algunos de los mecanismos habituales en la programación orientada a objetos.
- Es un **superset** de JavaScript.
- Se **compila** a JavaScript.
- Añade mecanismos como el tipado de variables, uso de clases, módulos e interfaces y nos permite detectar errores en tiempo de compilación.

## 3.2 EJEMPLO

```
// En JavaScript funciona perfectamente
// mientras que en TypeScript te dir
// á que no puedes asignar un número
// a una variable de tipo String.
let miString = 'Esto es un string';
miString = 4;
```

## 3.3 INSTALACIÓN

- Para poder instalar TypeScript necesitamos tener instalado NPM (el gestor de paquetes de NodeJS).
- Comprobamos que lo tenemos instalado y ejecutamos el siguiente comando para realizar la instalación de TypeScript:

```
$ npm install typescript -g
```

## 3.4 COMPILAR A JAVASCRIPT

- Para compilar nuestro código TypeScript a JavaScript escribimos el siguiente comando en la consola:

```
$ tsc archivo.ts
```

## 3.5 TSC

- TSC (TypeScript Compiler) es la herramienta que se encarga de compilar el código.
- Al compilar el archivo con el código en TypeScript se genera un archivo con extensión .js que contiene el código JavaScript equivalente al archivo que hemos compilado.
- El TSC incluye también los **watchers** que permiten vigilar los cambios en el archivo TypeScript, y cuando se realiza algún cambio se encarga de compilar el archivo automáticamente sin que tengamos que intervenir.

```
$ tsc -w archivo.ts
```

# 4 VARIABLES

## 4.1 TIPOS DE VARIABLES

Tipo	Descripción
number	Representa tanto números enteros como fracciones.
string	Representa una cadena de caracteres.
boolean	Representa valores lógicos (true o false).
void	Cuando una función no devuelve ningún tipo.
null	Representa la ausencia (intencionada) de valor.
undefined	Representa una variables sin inicializar.
any	Acepta cualquier tipo de los mencionados antes.

## 4.2 TUPLAS

- En Typescript se introduce un tipo de datos parecido a los Arrays, que son las **Tuplas**. Las tuplas son arrays de dos elementos, y para indicar que la variable es una tupla, se añaden los tipos de los dos elementos entre corchetes.

```
let telefono: [string, number] = ['+34', 637291043];
let direccion: [string, string] = ['Baker Street', '221B'];
```

- Para acceder a los valores de la tupla, hay que hacerlo como cuando se accede a un array, indicando la posición en la que se encuentra el valor que queremos obtener.

## 4.3 INICIALIZACIÓN (I)

- Al declarar una variable se le puede indicar el tipo de datos que va a guardar y el valor con el que se inicializa la variable.

```
// Declara una variable de un tipo y le asigna un valor.  
// let [identificador]: [tipo] = [valor];  
let miString: string = 'Esto es un string';
```

## 4.4 INICIALIZACIÓN (II)

- Al declarar la variable podemos **indicar el tipo de datos sin asignarle un valor** de inicialización.

```
// Declara una variable de un tipo, pero sin asignarle ningún valor.  
// let [identificador]: [tipo];  
let miNúmeroDeLaSuerte: number;
```

## 4.5 INICIALIZACIÓN (III)

- Si vamos a inicializar la variable cuando se declara, no hace falta indicarle de que tipo es ya que TypeScript infiere el tipo del valor que se le ha asignado.

```
// Declara una variable sin tipo a la que le asigna un valor.  
// El tipo se infiere del valor.  
// let [identificador] = [valor];  
let esVerdad = true;
```

## 4.6 INICIALIZACIÓN (IV)

- Por último podemos declarar una variable sin darle un valor ni indicarle de que tipo va a ser. En este caso la variable tendrá como tipo any y como valor undefined.

```
// Declara una variable sin tipo ni valor.  
// El tipo será any y el valor undefined.  
// let [identificador];  
let nombre;
```

## 4.7 INICIALIZACIÓN (V)

- Cuando una variable es de tipo any, esta puede recibir distintos tipos de valores sin que muestre errores.

```
// let cualquierValor;  
let cualquierValor: any;  
cualquierValor = true;  
cualquierValor = 'Cualquier valor';  
cualquierValor = 11;
```

## 4.8 UNIÓN DE TIPOS

- La unión de tipos nos permite indicar que una variable, un parámetro o el retorno de una función puede ser de varios tipos. Para usar la unión de tipos, a la hora de indicar el tipo de algún elemento hay que poner todos los tipos separados por |.

```
let unionType: number | string;  
unionType = 3;  
unionType = 'Y ahora un string';
```

## 4.9 ALIAS PARA TIPOS

- TypeScript nos permite crear **alias para los tipos**, es decir, que podremos crear nuestros propios tipos o cambiarle el nombre a los que ya existen.
- Para crear un tipo, se usa la palabra **type** seguida del alias que le queremos poner, y a esto se le iguala el tipo o tipos que queremos que represente.

```
type texto = string;
let unTexto: texto;
unString = 'Un string';

type miTipo = string | number;
let conAlias: miTipo;
conAlias = 'Un texto';
conAlias = 4;
```

## 4.10 TEMPLATE LITERALS (I)

- Los templates literals son una nueva forma de crear strings. Para usarlos hay que añadir el texto que queremos crear entre las comillas ``.

```
let texto = `Si eres bueno en algo, nunca lo hagas gratis`;
```

- El principal problema a la hora de crear strings a los que les queremos asignar valores que nos devuelve alguna expresión, es que tenemos que ir concatenándolos junto al texto que queremos mostrar, y esto puede darnos bastante trabajo si tenemos que mostrar bastantes valores almacenados en variables.

## 4.11 TEMPLATE LITERALS (II)

- Con los template literals vamos a evitar concatenar todo esto. Para ello, donde queramos mostrar una variable o el resultado de una expresión, vamos a añadir la expresión o la variable entre \${}.

```
let cuenta = '2+2';
let resultado = `El resultado de ${cuenta} es ${2+2}`;
```

## 4.12 TEMPLATE LITERALS (III)

- Por último, si queremos añadir saltos de línea, ahora lo podemos hacer de una forma mucho más sencilla que antes. Solo hay que saltar de línea dentro del **template literal** en lugar de añadir `\n` como haríamos usando el método antiguo.

```
let textoMultiLinea = `Este texto aparece  
en varias  
líneas`;
```

## **4.13 DESESTRUCTURACIÓN (I)**

- La desestructuración nos permite sacar de un array o un objeto los valores a unas variables sin necesidad de hacerlo uno a uno.

## 4.14 DESESTRUCTURACIÓN (II)

- En el caso de los objetos las variables a las que vamos a sacar el valor tienen que ir entre llaves y tener el mismo nombre que las propiedades que hay en el objeto.

```
let obj = {  
    nombre: 'Lucifer',  
    apellido: 'Morningstar'  
};  
let { nombre, apellido } = obj;
```

## 4.15 DESESTRUCTURACIÓN (III)

- Mientras que en los arrays, el nombre de las propiedades no importa cual sea, pero tienen que ir entre **corchetes**.

```
let arrayNums = [2, 5, 7];
let [n1, n2, n3] = arrayNums;
```

# **5 OPERADORES**

# 5.1 OPERADORES ARITMÉTICOS

Operador	Ejemplo
+ (Suma)	$5 + 10 = 15$
- (Resta)	$8 - 3 = 5$
* (Multiplicación)	$2 * 4 = 8$
/ (División)	$9 / 4 = 2.25$
% (Módulo)	$10 \% 3 = 1$
++ (Incremento)	<code>a=1; a++; =&gt; a=2;</code>
-- (Decremento)	<code>a=5; a--; =&gt; a=4;</code>

## 5.2 OPERADORES RELACIONALES

Operador	Ejemplo
< (Menor que)	$3 < 7$ (true)
> (Mayor que)	$8 > 9$ (false)
$\leq$ (Menor o igual que)	$2 \leq 2$ (true)
$\geq$ (Mayor o igual que)	$9 \geq 4$ (true)
$\equiv$ (Igual a)	$3 \equiv 2$ (false)
$\neq$ (Distinto de)	$4 \neq 5$ (true)

## 5.3 OPERADORES LÓGICOS

Operador	Ejemplo
&& (AND)	$10 > 3 \&& \text{false}$ (false)
(OR)	$10 > 3    \text{false}$ (true)
! (NOT)	$!(10 > 3)$ (false)

## 5.4 OPERADORES DE ASIGNACIÓN

- Para  $a = 5$  y  $b = 4$ :

Operador	Ejemplo
= (Igual)	$a = b (4)$
+= (Suma)	$a += b \Rightarrow a = a + b (9)$
-= (Resta)	$a -= b \Rightarrow a = a - b (1)$
*= (Multiplicación)	$a *= b \Rightarrow a = a * b (20)$
/= (División)	$a /= b \Rightarrow a = a / b (1.25)$

## 5.5 OPERADOR CONDICIONAL O TERNARIO (?)

- Este operador es equivalente al if...else.

```
// [condicion] ? [si es true] : [si es false]
let resultado = (5 < 3) ? 'Es menor' : 'Es mayor'; // resultado = 'Es mayor'
```

## 5.6 OPERADOR DE TIPO (TYPEOF)

- Este operador nos dice el tipo de una variable o valor.

```
// typeof [variable o valor]
let a = 6;
typeof a; // => number
```

# **6 ESTRUCTURAS CONDICIONALES**

## 6.1 IF

- La instrucción condicional **if** comprueba una condición, y si esta condición resulta ser **true**, entonces se ejecutará el bloque de código perteneciente al **if**. En caso de ser **false**, el bloque de código del **if** se salta y la ejecución sigue a partir de donde termina ese bloque.

```
if (nota >= 5) {  
    console.log('Has aprobado');  
}
```

## 6.2 IF - ELSE

- La instrucción condicional `if ... else` también va a comprobar una condición. En este caso, si la condición resulta ser `true` se va a ejecutar el bloque de código del `if`, mientras que si la condición es `false`, se ejecutará el bloque del `else`.

```
if (nota >= 5) {  
    console.log('Has aprobado');  
} else {  
    console.log('Has suspendido');  
}
```

## 6.3 IF - ELSE IF

- Si necesitamos tener en nuestro código más de un camino, podemos añadir más instrucciones **if** justo después de la instrucción del **else**. De esta forma, si la primera condición no se cumple, comprobará la del segundo **if** y así sucesivamente hasta el último **else**.

```
if (nota < 5) {  
    console.log('Has suspendido');  
} else if (nota < 6) {  
    console.log('Suficiente');  
} else if (nota < 8) {  
    console.log('Bien');  
} else if (nota < 10) {  
    console.log('Notable');  
} else {  
    console.log('Sobresaliente');  
}
```

## 6.4 SWITCH (I)

- La instrucción **switch** va a comparar el valor que recibe, con cada uno de los valores que hay en los casos (instrucción **case**), y va a ejecutar el bloque de código correspondiente al caso cuyo valor coincide con el valor que se le pasa al **switch**.
- El **switch** tiene un caso especial (**default**) cuyo bloque de código se va a ejecutar cuando ninguno de los casos coincide con el valor que se está comparando.
- Al final de cada **case**, se pone la palabra instrucción **break** que le dice al interprete de JavaScript que ya ha terminado de ejecutar el **switch** y que puede seguir ejecutando el código a partir de su bloque.

## 6.5 SWITCH (II)

```
switch (dia) {  
    case 'Sábado':  
        console.log('Fin de semana');  
        break;  
    case 'Domingo':  
        console.log('Fin de semana');  
        break;  
    default:  
        console.log('Entre semana');  
        break;  
}
```

## 6.6 BUCLES

- Los bucles van a comprobar una condición. Si esta condición es **true**, el bloque de código se va a ejecutar. Una vez ejecutado el bloque de código, se vuelve a comprobar la condición y si sigue siendo **true** se vuelve a ejecutar otra vez, y así sucesivamente hasta que la condición sea **false** que será cuando deje de ejecutar ese bloque de código correspondiente al bucle.

## 6.7 FOR

- El **for** se usa para cuando necesitamos ejecutar un bloque de código un número de veces conocido. Este el bucle más común a la hora de usarse. En este bucle la condición es el número de veces que se tiene que repetir el bloque de código. El valor que se usa en la condición tendremos que inicializarlo, e ir incrementandolo/decrementandolo para que no se quede en un bucle infinito al final de la ejecución del bloque.

```
for (let i = 0; i < 5; i++) {  
  console.log('Iteración ' + (i + 1));  
}
```

## 6.8 FOR IN

- El **for in** es una versión del bucle **for** que se usa para iterar sobre Arrays u Objetos. En cada iteración se va a ir guardando en la variable la **posición** y si se está usando para iterar sobre un objeto, se va a guardar la **clave**.

```
let nums = [1, 2, 3, 4];
for (let j in nums) {
  console.log('Posición del array en esta iteración' + j);
}
```

## 6.9 FOR OF

- El **for of** es como el bucle **for in** que hemos visto anteriormente, solo que esta vez en lugar de guardar en la variable la posición o la clave, se guarda el **valor** del elemento.

```
let nums = [1, 2, 3, 4];
for (let k of nums) {
  console.log('Elemento del array en esta iteración' + k);
}
```

## 6.10 WHILE

- En caso de no conocer el número de veces que se tiene que ejecutar el bloque de código, deberíamos usar el `while`, en el que la condición puede ser otra expresión que no sea un contador (como en el `for`) y el código se repetirá hasta que esta condición sea `false`.

```
let num = prompt('Dame un número... Introduce -1 si quieras terminar.');
while (num != -1) {
    console.log('Has introducido el número ' + num);
    num = prompt('Dame un número... Introduce -1 si quieres terminar.');
}
```

## 6.11 DO - WHILE

- Este bucle es muy parecido al `while`. La única diferencia entre ellos, es que el bloque de código correspondiente a este bucle, se va a ejecutar la primera vez aunque la condición sea `false`. En este caso, primero se ejecuta el bloque, luego comprueba la condición, y a partir de aquí todo funciona igual que el `while`.

```
do {  
  let num = prompt('Dame un número... Introduce -1 si quieres terminar.');//  
  console.log('Has introducido el número ' + num);  
} while (num != -1);
```

# 7 FUNCIONES

## 7.1 INTODUCCIÓN

- Las funciones son bloques de código mantenible y reusable que realizan una tarea específica.
- Las funciones nos ayudan a organizar el programa en bloques de código que nos facilita su lectura.
- Para ejecutar una función solo hace falta llamarla donde quieras ejecutar el bloque de código que contiene.
- A las funciones se les pueden pasar parámetros y algunas devuelven valores.

## 7.2 TYPESCRIPT

- Son **tipadas**, es decir que le vamos a poder indicar de que tipo son los parámetros que recibe la función, y que tipo de datos tiene que devolver.
- Permite **sobrecarga de funciones** (funciones con el mismo nombre y distintos parámetros).
- Los parámetros pueden ser **requeridos u opcionales**.
- Soporta **arrow functions**.
- Se le pueden poner **valores por defecto a los parámetros**.
- Soporta **parámetros rest**.

## 7.3 JAVASCRIPT

- No tiene tipos.
- No se permite sobrecargar funciones (funciones con el mismo nombre y distintos parámetros).
- Todos los parámetros son opcionales.
- Soporta arrow functions en ES6.
- Se le pueden poner valores por defecto a los parámetros en ES6.
- Soporta parámetros rest en ES6.

## 7.4 TIPOS EN PARÁMETROS Y EN VALORES DE RETORNO

- En las funciones es posible indicar el tipo de los parámetros que va a recibir, y el tipo del valor que va a devolver dicha función.

```
function crearUsuarioId(nombre: string, id: number): string {  
    return nombre + id;  
}
```

## 7.5 ARROW FUNCTIONS (I)

- Las arrow functions que a veces se les llama **funciones lambda**, son funciones anónimas que se usan muy amenudo tanto en TypeScript como en JavaScript.
- Las arrow functions nos ahorrarán el tener que escribir código y se llaman arrow functions porque el cuerpo y los parámetros van separados por una flecha (`=>`).

## 7.6 ARROW FUNCTIONS (II)

```
let misPeliculas = [
  { titulo: 'Scary movie', genero: 'comedia'},
  { titulo: 'La jungla de cristal', genero: 'accion'},
  { titulo: 'Los mercenarios', genero: 'accion'},
  { titulo: 'Salvar al soldado Ryan', genero: 'belica'}
];

let peliculasComedia = misPeliculas.filter(function(pelicula) {
  return pelicula.genero == 'comedia';
});

let peliculasAccion = misPeliculas.filter(pelicula =>
  pelicula.genero == 'accion');
```

## 7.7 ARROW FUNCTIONS (III)

- Cuando estas funciones no reciben parámetros o reciben más de uno, entonces hay que ponerlos entre **paréntesis**, mientras que si reciben solo uno, da igual si ponemos los paréntesis o no.
- Y en caso de tener más de una linea de código, el cuerpo de la función tiene que ir entre **llaves**.

```
misPeliculas.forEach(() => console.log('Vista!'));
misPeliculas.forEach(pelicula => console.log(pelicula.titulo + ' vista!'));
misPeliculas.forEach((pelicula, index) => console.log(pelicula.titulo + ' (' +
misPeliculas.forEach((pelicula, index) => {
  let texto = pelicula.titulo + ' (' + pelicula.genero + ')' + ' vista!';
  console.log(texto);
});
```

## 7.8 ARROW FUNCTIONS (IV)

- Normalmente cuando necesitamos usar la variable `this` dentro de un `callback`, tenemos que asignarle su valor a otra variable que se suele llamar `self`. Con las arrow functions no hace falta capturar la variable `this` sino que te deja usarla directamente.

```
function pelicula() {  
  let self = this;  
  self.añoEstreno = 2000;  
  setTimeout(function() {  
    console.log(self.añoEstreno);  
  }, 1500);  
}  
  
function peliculaArrow() {  
  this.añoEstreno = 2000;  
  setTimeout(() => {  
    console.log(this.añoEstreno);  
  }, 1500);  
}
```

## 7.9 DEFINICIÓN DE FUNCIONES

- Podemos indicar la forma que va a tener una función.

```
let generadorIds1: (chars: string, nums: number) => string;
let generadorIds2: (chars: string, nums: number) => string;

function crearUsuarioId2(nombre: string, id: number): string {
    return nombre + id;
}

generadorIds1 = crearUsuarioId2;
let miId1 = generadorIds1('angel', 201);
console.log(miId1);

generadorIds2 = (nombre: string, id: number) => { return id + nombre; };
let miId2 = generadorIds2('pedro', 104);
console.log(miId2);
```

## 7.10 PARÁMETROS OPCIONALES Y POR DEFECTO

- Podemos indicar que un parámetro es opcional poniéndole ? después del nombre del parámetro.
- Todos los parámetros opcionales deben de ir situados después de todos aquellos que son obligatorios.
- Para asignarle un valor por defecto le igualamos el valor en la sección donde se definen los parámetros.

```
function muestraDatosUsuario(nombre: string = 'Blanca', email?: string): void {
    ...
}

muestraDatosUsuario('James', 'jmcgill@bettercallsaul.com');
muestraDatosUsuario('Robb');
muestraDatosUsuario();
```

## 7.11 PARÁMETROS REST

- Los parámetros rest, se usan para **recoger un conjunto de valores en un array**.

```
function getNumeroLoteria(...nums: number[]): string {  
  return nums.join(', ');  
}  
  
let num = getNumeroLoteria(1, 5, 12, 22, 35, 37);  
console.log(num);
```

## 7.12 SOBRECARGA DE FUNCIONES

- Con la sobrecarga de funciones vamos a tener la posibilidad de **crear varias funciones con el mismo nombre**, pero con distintos parámetros.

```
function hola(valor: number) {  
    console.log('Hola número '+ valor)  
}  
function hola(valor: string) {  
    console.log('Hola texto '+ valor)  
}
```

# **8 INTERFACES**

## 8.1 INTRODUCCIÓN

- Las interfaces son **contratos** que definen como va a ser un tipo de dato.
- Estas interfaces no se compilan a nada en JavaScript.
- Es una colección de **definiciones de propiedades y métodos** que no se llegan a implementar.
- Solo indica de que tipo tienen que ser las propiedades y en cuanto a los métodos, define que parámetros recibe y que tiene que devolver.

## 8.2 DEFINIR UNA INTERFAZ

Una interfaz se define de la siguiente forma:

```
interface Pelicula {  
    titulo: string,  
    duracion?: number, // Parámetro opcional  
    ganadoraOscar?: (gana: boolean) => void // Los métodos sin implementar  
}  
  
let pelicula: Pelicula = {  
    titulo: 'Los mercenarios',  
    duracion: 113,  
    ganadoraOscar: (gana: boolean) =>  
        console.log(gana ? 'Ha ganado un oscar' : 'No ha ganado ningún oscar')  
}  
  
pelicula.ganadoraOscar(true);
```

## 8.3 EXTENDIENDO INTERFACES

- Las interfaces pueden ser extendidas con la palabra reservada **extends**.

```
interface Persona {  
    nombre: string,  
    email: string  
}  
  
interface DirectorCine extends Persona {  
    numPeliculasDirigidas: number  
}  
  
let director: DirectorCine = {  
    nombre: 'Joss Whedon',  
    email: 'jossw@gmail.com',  
    numPeliculasDirigidas: 20  
}
```

## 8.4 TIPOS DE CLASES

- Para indicar que una clase implementa una interfaz se usa la palabra reservada **implements**.

```
interface Desarrollador {  
    trabaja: () => void;  
}  
  
class DesarrolladorJavascript implements Desarrollador {  
    trabaja() {  
        console.log('Desarrollo aplicaciones con JavaScript');  
    }  
}  
  
let desarrollador: Desarrollador = new DesarrolladorJavascript();  
desarrollador.trabaja();
```

**9 CLASES**

## 9.1 INTRODUCCIÓN

- Las clases son plantillas que nos van a permitir crear objetos.
- Todos los objetos que se crean con las clases van a tener las mismas propiedades y los mismos métodos.
- Las clases encapsulan una funcionalidad que se puede reutilizar.

## 9.2 CONSTRUCTORES

- Se encargan de crear nuevas instancias de una clase.
- Son métodos cuyo nombre es **constructor** y solo puede haber uno por clase.
- Pueden recibir parámetros que van a servir para darle valores a las propiedades de los objetos.
- Para crear objetos se usa la palabra **new** seguida del nombre de la clase con los parámetros.

```
class Mascota {  
  constructor(nombre: string, tipo: string) { ... }  
}  
  
let perro = new Mascota('Rocky', 'perro');
```

## 9.3 PROPIEDADES Y MÉTODOS (I)

- Al igual que las interfaces, las clases pueden tener propiedades y métodos, pero en este caso, si que se van a **implementar**.

```
class Mascota {  
    public nombre: string;  
    public tipo: string;  
  
    constructor(nombre: string, tipo: string, sonido: string) {  
        this.nombre = nombre;  
        this.tipo = tipo;  
    }  
  
    toString (): void {  
        console.log(`Mi ${this.tipo} se llama ${this.nombre}`);  
    }  
}  
  
let perro = new Mascota('Rocky', 'perro');  
perro.toString();
```

## 9.4 PROPIEDADES Y MÉTODOS (II)

- Hay dos formas de inicializar las propiedades, una es declarando la propiedad y en el constructor le asignamos el valor.
- Y la otra es **indicando la visibilidad de la propiedad en el propio constructor**.

```
class Mascota {  
    constructor(public nombre: string, public tipo: string) {}  
  
    toString (): void {  
        console.log(`Mi ${this.tipo} se llama ${this.nombre}`);  
    }  
}  
  
let perro = new Mascota('Rocky', 'perro', 'guau');  
perro.toString();
```

## 9.5 PROPIEDADES Y MÉTODOS (III)

- También podemos tener propiedades **estáticas**, que no son propiedades de la instancia, sino que son propiedades de la clase.
- Para acceder al valor de esa propiedad tenemos que usar el nombre de la clase en lugar de la instancia de un objeto.

```
class Coche {  
    constructor(public marca: string) { }  
    static numRuedas: number = 4;  
}  
  
let coche = new Coche('Seat');  
let marca = coche.marca;  
let ruedas = Coche.numRuedas;
```

## 9.6 HERENCIA (I)

- La herencia es un medio por el que una clase puede compartir con sus subclases las propiedades y métodos.
- Si tenemos una clase que tiene dos clases hijas o subclases, estas dos subsclases podrán usar la funcionalidad que tiene la clase padre.
- Para indicar que una clase hereda de otra se usa la palabra **extends**.
- En caso de que una subclase tenga un método con el mismo nombre que un método del padre, esta usará su propio método, en caso de que no, entonces usará el método del padre.

## 9.7 HERENCIA (II)

```
class Persona {  
    constructor(public nombre: string, public dni: string) { }  
    toString() {  
        console.log(`Nombre: ${this.nombre} DNI: ${this.dni}`);  
    }  
}  
class Alumno extends Persona {  
    constructor(nombre: string, dni: string,  
               public numMatricula: number, public creditosAprobados: number) {  
        super(nombre, dni);  
    }  
    toString() {  
        super.toString();  
        console.log(`Num. Matrícula: ${this.numMatricula}  
                   Creditos Aprobados: ${this.creditosAprobados}`);  
    }  
}
```

## 9.8 VISIBILIDAD

- En TypeScript nos podemos encontrar los tres tipos de visibilidad que hay en otros lenguajes: **public**, **private** y **protected**.
- Por defecto todas las propiedades y los métodos de una clase van a ser **públicos** a no ser que se les indique otro tipo de visibilidad. Al ser públicos, se van a poder acceder a ellos tanto desde dentro de la clase, como fuera de esta (a través de los objetos).

## 9.9 PUBLIC

- Si queremos indicar de forma explícita que una propiedad o método va a ser público, hay que añadir delante del nombre la palabra reservada **public**.

```
class PersonaPublica {  
    constructor(public nombre: string) { }  
    toString () {  
        console.log(`Me llamo ${this.nombre}`);  
    }  
}
```

## 9.10 PRIVATE

- Para indicar que una propiedad o método es **privado** hay que añadir delante del nombre la palabra reservada **private**. Y en este caso a esta propiedad o método solo se va a poder acceder desde dentro de la clase.

```
class PersonaPrivada {  
    constructor(private nombre: string) { }  
    toString () {  
        console.log(`Me llamo ${this.nombre}`);  
    }  
}
```

## 9.11 PROTECTED

- Y por último, aunque en JavaScript no existe, también vamos a poder indicar que un método o propiedad tiene visibilidad protegida añadiendo la palabra **protected** delante del nombre. Y este tipo de visibilidad actuá como la privada, pero con la diferencia de que una subclase si que puede acceder a los métodos y propiedades protegidas de la clase de la que hereda.

```
class PersonaProtegida {  
    constructor(protected nombre: string) { }  
    toString() {  
        console.log(`Me llamo ${this.nombre}`);  
    }  
}
```

## 9.12 READONLY

- En TypeScript también podemos indicar que las propiedades sean solo de **lectura**, por lo que no se van a poder modificar, son como constantes de la clase.
- Estas propiedades llevan la palabra reservada **readonly** delante del nombre y se tienen que inicializar en el constructor o en la declaración.

```
class PersonaSoloLectura {  
    constructor(readonly nombre: string) { }  
    toString() {  
        console.log(`Me llamo ${this.nombre}`);  
    }  
}
```

## 9.13 GETTERS Y SETTERS

- TypeScript permite usar **getters** y **setters** para interceptar los accesos a las propiedades de las clases. De esta forma se puede tener un mayor control sobre quien accede o modifica estas propiedades añadiendo comprobaciones dentro de estos métodos.

## 9.14 GETTERS

- Los **getters** nos van a devolver el valor de la propiedad y se declaran añadiendo la palabra **get** delante del nombre de la función que se va a ejecutar cuando se acceda a la propiedad correspondiente al getter.

```
class Persona {  
    constructor(private _nombre: string) { }  
    get nombre(): string {  
        return this._nombre;  
    }  
    toString() {  
        console.log(`Me llamo ${this.nombre}`);  
    }  
}
```

## 9.15 SETTERS

- Los **setters** por otro lado se van a encargar de modificar el valor de la propiedad y se declaran usando la palabra **set** delante del nombre de la función que se va a ejecutar cuando se vaya a igualar un valor a la propiedad correspondiente al setter.

```
class Persona {  
    constructor(private _nombre: string) { }  
    set nombre(nombre: string) {  
        this._nombre = nombre;  
    }  
    toString() {  
        console.log(`Me llamo ${this.nombre}`);  
    }  
}
```

## 9.16 BUENAS PRÁCTICAS

- Como buena práctica, las propiedades privadas se declaran con un **guión bajo** al inicio del nombre. Y a los **getters** y **setters** se les da el mismo nombre sin el **guión bajo**.
- Ahora para usar el **getter** y el **setter** solo hay que llamar al método **nombre**. Si se le asigna un valor se ejecutará el **setter**, y si solo se pide el valor entonces se ejecuta el **getter**.

```
let p = new Persona('Chloe');

p.nombre = 'Maze'; // Se llama al setter

p.toString(); // Se llama al getter
```

## 9.17 CLASES ABSTRACTAS (I)

- Son clases que van a ser heredadas y que implementan métodos que van a compartir las clases que heredan de esta.
- Las clases abstractas llevan la palabra **abstract** y no se pueden instanciar.

```
abstract class Animal {  
    constructor(protected _nombre: string) { }  
    get nombre() {  
        return this._nombre;  
    }  
    set nombre(nombre: string) {  
        this._nombre = nombre;  
    }  
}  
  
class Gato extends Animal {  
    constructor(nombre: string) {  
        super(nombre);  
    }  
}
```

## 9.18 CLASES ABSTRACTAS (II)

- Pueden declarar **métodos abstractos** que serán implementados en las clases que heredan de ella.
- Estos métodos llevan la palabra **abstract** y no hay que implementarlos.

```
abstract class Animal {  
    constructor(protected _nombre: string) { }  
  
    get nombre() {  
        return this._nombre;  
    }  
  
    set nombre(nombre: string) {  
        this._nombre = nombre;  
    }  
  
    abstract hazSonido(): void;  
}
```

# **10 ENUMERADOS**

## 10.1 INTRODUCCIÓN

- Los enumerados nos permiten restringir el contenido de una variable a una serie de **valores predefinidos**. A cada uno de estos valores se les asigna por defecto un valor numérico que empieza en 0.

## 10.2 CREACIÓN

- Para crear un enumerado, se usa la palabra **enum** seguida del nombre del enumerado y todos los valores entre llaves {}.

```
enum Direction { Up, Down, Left, Right };
console.log(Direction.Up);
```

## 10.3 PERSONALIZACIÓN

- Podemos establecer estos valores nosotros mismos igualándolos a lo que queramos.
- En caso de ponerle un número al primer valor, el resto obtendrán el valor anterior incrementado en una unidad.
- También les podemos dar a cada uno el valor que queramos.

```
enum Category { Fantasy = 1, Comedy, Action, Science_Fiction };  
console.log(Category.Comedy);  
enum Languages { Spanish = 2, French = 8, English = 4, Germany = 7 };  
console.log(Languages.English);  
console.log(Languages[8]);
```

**11 GENÉRICOS**

## 11.1 INTRODUCCIÓN

- Los genéricos son bloques de código que nos permiten trabajar con múltiples tipos de datos.
- Usando los genéricos vamos a poder **crear versiones genéricas** de funciones, clases e interfaces.
- Los parámetros que se le pasan a los genéricos van a indicar sobre qué tipo de datos se van a realizar operaciones.

```
let array0fNumbers: Array<number>;
array0fNumbers = [1, 2, 3, 4];
```

## 11.2 FUNCIONES GENÉRICAS

- Podemos usar los genéricos con las funciones.

```
function dameItemAleatorio<T>(items: Array<T>): T {  
  let posicion = Math.floor(Math.random() ** items.length);  
  return items[posicion];  
}  
  
let itemNum = dameItemAleatorio<number>([1, 3, 5, 2]);  
let itemFamiliaGot = dameItemAleatorio<string>(  
  ['Stark', 'Lannister', 'Baratheon', 'Targaryen', 'Martell', 'Greyjoy']);
```

# 11.3 INTERFACES Y CLASES GENÉRICAS

- Se pueden crear interfaces y clases con tipos genéricos.

```
interface inventario<T> {
    addItem: (item: T) => void;
}
interface Portatil {
    marca: string;
}
class Catalogo<T> implements inventario<T> {
    private catalogo = new Array<T>();
    addItem(item: T) {
        this.catalogo.push(item);
    }
}
let catalogoPortatil = new Catalogo<Portatil>();
catalogoPortatil.addItem({marca: 'HP'});
catalogoPortatil.addItem({marca: 'Lenovo'});
```

**12 MÓDULOS**

## 12.1 INTRODUCCIÓN

- Los módulos nos permiten agrupar bloques de código y exportarlos para poder usarlos en cualquier lugar de la aplicación cuando queramos.
- Todo aquello que hemos puesto en un módulo solo va a poder ser usado en otros módulos si se está exportando, por lo tanto tendremos que indicar, que cosas hay que exportar con la palabra **export**.

## 12.2 EXPORT

```
export class Mascota {  
    constructor(public nombre: string, public tipo: string) {}  
}
```

```
class Mascota {  
    constructor(public nombre: string, public tipo: string) {}  
}  
export { Mascota };
```

## 12.3 IMPORT

- Para poder usar aquello que se está exportando desde un módulo, tendremos que importarlo en el archivo donde lo necesitemos usando la palabra **import** seguida de lo que se quiere importar y indicando desde que archivo hay que importarlo. A la hora de importarlo le podemos poner un alias a lo que se importa.

```
import { Mascota as Masc } from './mascota';  
  
let perro = new Masc('Toby', 'perro');
```

## 12.4 IMPORT \*

- Hay otra forma de importar módulos y es usando el asterisco, que importará todo aquello que se está exportando. A la hora de usar esta forma tendremos que darle un alias para poder acceder a las cosas que se están importando.

```
import * as masc from './mascota';

let perro = new masc.Mascota('Toby', 'perro');
```

## 12.5 EXPORTANDO POR DEFECTO

- Hay otra opción para exportar a parte de las vistas anteriores, y es la exportación por defecto que se usa cuando vamos a exportar solo una cosa del módulo. Aquí no hace falta ponerle nombre a lo que estamos exportando.

```
export default class {
  constructor(public nombre: string, public tipo: string);
}
```

```
import Mascota from './mascota';

let perro = new Mascota('Toby', 'perro');
```

# **13 NAMESPACES**

## 13.1 INTRODUCCIÓN

- Los namespaces nos permiten separar el código en distintos scopes para no ensuciar el scope global (window).
- Cuando la aplicación se vuelve demasiado grande, podemos cometer errores como duplicar el nombre de alguna función, clase... y esto puede provocar errores en nuestra aplicación.
- Todo aquello que va dentro del namespace no va a interferir con lo que hay en el scope global, y de esta forma tendremos mucho más limpio nuestro código y evitaremos algunos errores comunes en proyectos grandes.

## 13.2 CREACIÓN

- Para crear un namespace, se pone la palabra reservada **namespace** seguida del nombre que le vamos a dar, y entre llaves todo el código correspondiente a ese namespace (constantes, funciones, clases ...).
- Todo aquello a lo que se necesite acceder desde fuera del namespace se tiene que exportar.

## 13.3 EJEMPLO

```
namespace Validator {  
    const letras = /^w[^d_.]+$/g;  
    const numeros = /^d+$/g;  
    export function soloLetras(cadena: any) {  
        return letras.test(cadena);  
    }  
    export function soloNumeros(cadena: any) {  
        return numeros.test(cadena);  
    }  
}  
console.log(Validator.soloLetras('h0la'));  
console.log(Validator.soloNumeros(123));
```

# **14 PROMESAS**

## 14.1 INTRODUCCIÓN

- Las promesas son objetos que se usan en las operaciones asíncronas, por lo que no se sabe si vamos a obtener el resultado de la operación ahora, en el futuro o nunca.
- Para usar las promesas, hay que usar una versión de Javascript >= ES6
- Surgen sobre todo para mejorar la legibilidad de nuestro código y evitar tener que pasar el contenido de las funciones directamente como argumentos a nuestra llamada (el **callback hell**).

## 14.2 FUNCIONAMIENTO

- Al constructor de la promesa se le pasa una función que se encargará de realizar algunas operaciones asíncronas y de avisar si estas operaciones han terminado bien o ha ocurrido algún error.
- Esta función recibe dos parámetros que son los encargados de indicar que la promesa ha terminado correctamente (primer parámetro `resolve`) o que ha finalizado con algún error (segundo parámetro `reject`).

## 14.3 RESOLVE Y REJECT

- En caso de que haya ido bien la promesa, se usará la función `resolve` y se le pasará como parámetro los datos que queremos pasar al código que va a procesarlos.
- Mientras que si la promesa termina con algún error, entonces vamos a llamar a la función de `reject` pasandole la razón por la cual no se ha podido obtener los datos esperados.

## 14.4 CREACIÓN

```
let promesa: Promise<Array<number>> =  
  new Promise<Array<number>>(getNumeros);  
  
function getNumeros(resolve, reject) {  
  funcionAsincrona((arrayResultados)=>{  
    if (arrayResultados.length > 0) {  
      resolve(arrayResultados);  
    } else {  
      reject(new Error('Error'));  
    }  
  });  
}
```

## 14.5 ARROW FUNCTION

- Aunque en el ejemplo de arriba la función que recibe la promesa se ha declarado fuera, en la mayoría de los sitios nos encontraremos que a la promesa se le pasa una arrow function.

```
let promesa: Promise<Array<number>> =  
  new Promise<Array<number>>((resolve, reject) {  
    funcionAsincrona(arrayResultados)=>{  
      if (arrayResultados.length > 0) {  
        resolve(arrayResultados);  
      } else {  
        reject(new Error('Error'));  
      }  
    } );  
});
```

## 14.6 USO

- Una vez que tenemos la promesa construida, tenemos que procesar los resultados, y para ello se usan los métodos **then** (la promesa se ha terminado correctamente) y **catch** (la promesa ha terminado con algún error).

```
promesa.then((nums) => {
  console.log(nums);
}).catch((error) => {
  console.log('Error: ' + error.message);
});
```

## 14.7 PROMESAS ENCADENADAS

- Las promesas se pueden encadenar una detrás de otra poniendo los `then` seguidos, de esta forma podemos hacer que todas las peticiones asíncronas se ejecuten en el orden en que nosotros queremos que se hagan.

# **15 DECORADORES**

## 15.1 INTRODUCCIÓN

- Con la introducción de las clases existen ciertos escenarios que requieren características adicionales para soportar la anotación o modificación de clases y miembros de la clase. Los decoradores proporcionan una forma de añadir anotaciones y una sintaxis de metaprogramación para las declaraciones de clase y los miembros.

## 15.2 SOPORTE

- Los decoradores son una **característica experimental** que puede cambiar en futuras versiones.
- Para habilitar el soporte experimental para decoradores, debe habilitar la opción del compilador **experimentalDecorators** en la línea de comandos o en su `tsconfig.json`:

## 15.3 HABILITAR

Línea de comandos:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}
```

## 15.4 DEFINICIÓN

- Es un tipo especial de declaración que se puede adjuntar a una declaración de clase, método, getter/setter, propiedad o parámetro. Los decoradores utilizan la forma `@expresion`, donde la expresión debe evaluar una función que será llamada en tiempo de ejecución con información sobre la declaración decorada.

## 15.5 COMPOSICIÓN DEL DECORADOR

- Se pueden aplicar varios decoradores a una declaración, como en los siguientes ejemplos:

En una sola línea:

```
@f @g x
```

En múltiples líneas:

```
@f  
@g  
x
```

# 15.6 ORDEN DE EJECUCIÓN (I)

```
function f() {
    console.log("f(): evaluated");
    return function(destino, propiedadKey: string,
        descriptor: PropertyDescriptor) {
        console.log("f(): llamado");
    }
}

function g() {
    console.log("g(): evaluated");
    return function(destino, propiedadKey: string,
        descriptor: PropertyDescriptor) {
        console.log("g(): llamado");
    }
}
```

## 15.7 ORDEN DE EJECUCIÓN (II)

```
clase C {  
    @f()  
    @g()  
    method() {}  
}  
  
f(): evaluado  
g(): evaluado  
g(): llamado  
f(): llamado
```

## 15.8 TIPOS DE DECORADORES

- **De clase:** es declarado justo antes de una declaración de clase.
- **De método:** se declara justo antes de una declaración de métodos.
- **De getters y setters:** se declara justo antes de una declaración de getters o setters. TypeScript no permite decorar los getters y setters por separado.
- **De propiedades:** es declarado justo antes de una declaración de propiedad.
- **De parámetro:** se declara justo antes de una declaración de parámetros.

# **16 COMPILADOR**

## 16.1 INTRODUCCIÓN

- Hemos visto como compilar los archivos de TypeScript usando la línea de comandos, pero si queremos añadir varias opciones, estos comandos se van a hacer muy largos.
- En lugar de poner las opciones en la linea de comandos, podemos añadirlas en un fichero de configuración `tsconfig.json`.

## 16.2 INICIALIZACIÓN

- Este fichero de configuración se puede crear con el siguiente comando:

```
$ tsc --init
```

## 16.3 POR DEFECTO

- Una vez lanzado el comando, se debe de haber creado un archivo `tsconfig.json` con las siguientes opciones de configuración por defecto.

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es5",  
    "noImplicitAny": false,  
    "sourceMap": false  
  }  
}
```

## 16.4 OPCIONES

Opción	Descripción
compilerOptions	Opciones de compilación.
files	Archivos a compilar.
include	Expresión regular con los ficheros a incluir.
exclude	Expresión regular con los ficheros a excluir.

## 16.5 EJEMPLO

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es5",  
    "noImplicitAny": false,  
    "sourceMap": false,  
  },  
  "exclude": [  
    "node_modules",  
    "**/*.*spec.ts"  
  ],  
  "files": [  
    "app.ts"  
  ]  
}
```

## 16.6 COMPILEROPTIONS

- En el objeto de **compilerOptions** se pueden añadir distintas opciones que nos ayudarán a escribir mejor nuestro código. Todas estas opciones se pueden encontrar en <https://www.typescriptlang.org/docs/handbook/compiler-options.html>

## **16.7 SOURCERMAP**

- Esto nos permite tener archivos .map para debuguear.

## 16.8 NOIMPLICITANY

- En algunos casos como con los argumentos de las funciones, estos no pueden inferir el tipo de datos que van a guardar, por lo tanto puede no quedar claro que argumentos acepta para que este método funcione correctamente. Para evitar esto se puede activar la opción de `nolmplicitAny` que nos dará errores cuando esto ocurra.

## 16.9 NOEMITONERROR

- Cuando compilamos los archivos de TypeScript, en la consola pueden aparecernos errores, pero aun así el archivo se va a compilar a JavaScript al cual estos errores le dan igual porque va a funcionar correctamente. Podemos hacer que solo se compilen los archivos una vez que estos ya no tengan errores para asegurarnos que el código funcionará correctamente. Para ello vamos a añadir la opción de `noEmitOnError`.

## **16.10 PRETTY**

- Podemos hacer que los mensajes que aparecen por consola a la hora de compilar los archivos, salgan con colores para que sean más fáciles de leer con la opción pretty.

## **16.11 REMOVECOMMENTS**

- También se pueden eliminar todos los comentarios que hemos puesto en el código con la opción de `removeComments`.

## **16.12 OUTDIR**

- En caso de querer crear los archivos JavaScript compilados en una carpeta, se le puede indicar con la opción `outDir`.

## 16.13 WATCH

- Para evitar tener que estar lanzando constantemente el comando que compila los archivos, podemos añadir la opción `watch`, y de esta forma cada que un archivo cambie, se complilaran otra vez.

## 16.14 STRICTNULLCHECKS

- Otra de las opciones que pueden ayudarnos a tener una aplicación con menos errores es `strictNullChecks` que nos va a dar un error en caso de que una variable se esté usando sin llegarla a inicializar como en el siguiente código.

## 16.15 NOUNUSED

- En el código hay veces que podemos añadir variables locales o parámetros que luego no necesitamos y se nos olvida de quitar. Para evitar esto y que al compilar nos avise de que esos parámetros o variables no se están usando, se pueden usar las opciones `noUnusedParameters` y `noUnusedLocals`.

## 16.16 NOIMPLICITRETURNS

- Y hay otras veces que escribimos código que puede ejecutarse o no dependiendo del camino que se siga según las expresiones condicionales, y puede ser que en algún caso se nos olvide añadir un `return` por lo tanto uno de esos caminos no va a devolver ningún valor cuando se espera que si lo haga. En este caso podemos añadir la opción `nolImplicitReturns` para que nos avise si eso ocurre.

## 16.17 EJEMPLO

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es5",  
    "noImplicitAny": true,  
    "sourceMap": true,  
    "noEmitOnError": true,  
    "pretty": true,  
    "removeComments": true,  
    "outDir": "./js",  
    "watch": false,  
    "strictNullChecks": true,  
    "noUnusedParameters": true,  
    "noUnusedLocals": true,  
    "noImplicitReturns": true  
  }  
}
```