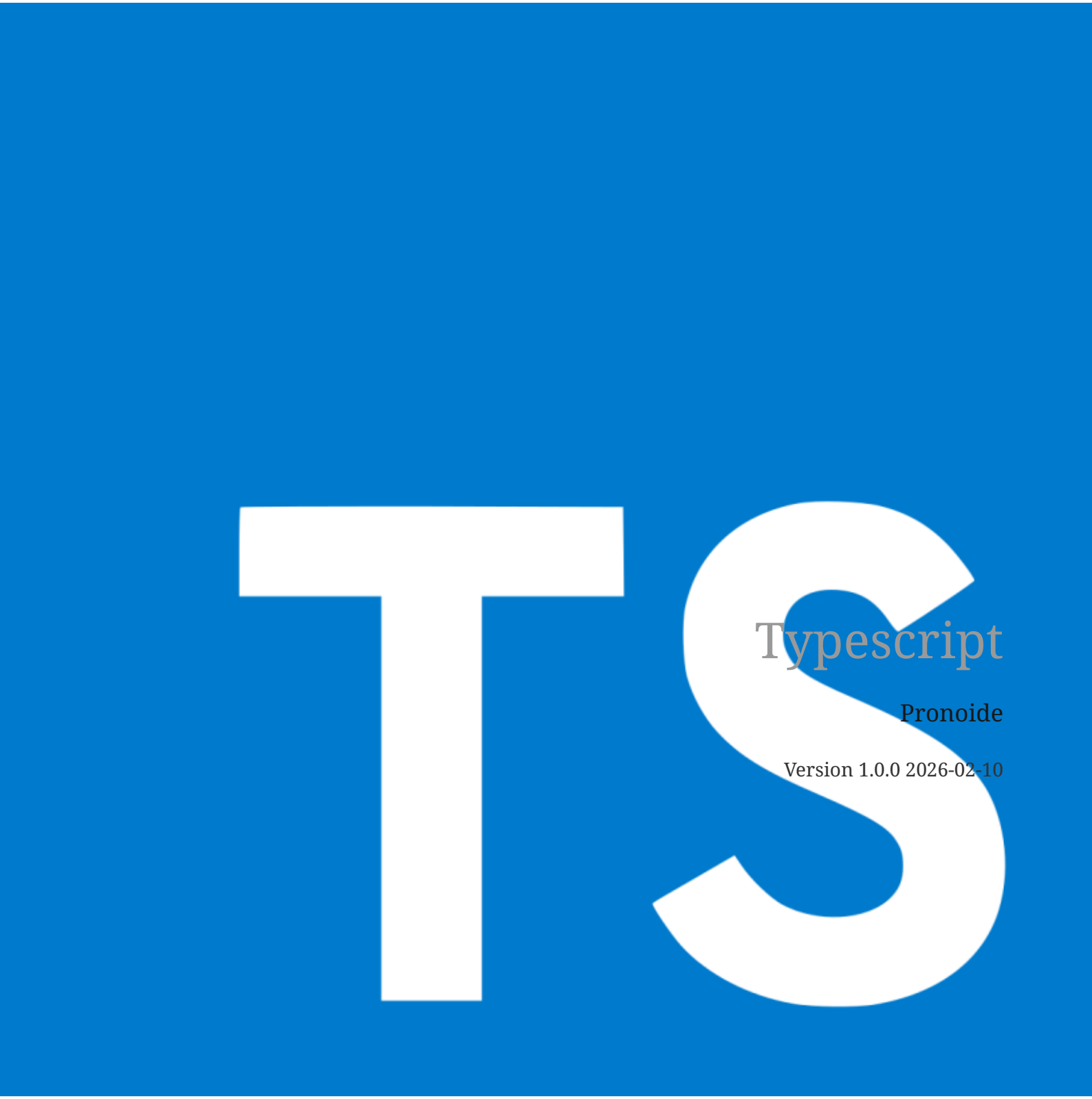


Typescript

Pronoide

Version 1.0.0 2026-02-10

Contenidos

Chapter 1. Introducción

TypeScript es un 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.

TypeScript es un superset de JavaScript, es decir, que se compila a JavaScript. Al final el navegador lo que ejecuta es JavaScript, y no sabrá que el código original estaba escrito en TypeScript.

Usamos TypeScript en lugar de JavaScript, porque añade mecanismos como el tipado de variables, uso de clases, módulos e interfaces y nos permite detectar errores en tiempo de compilación.

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

Chapter 2. Requisitos previos

Para poder empezar a trabajar con TypeScript, necesitaremos tener instalado lo siguiente:

- Node: link::https://nodejs.org/en/[https://nodejs.org/en/] (Recomendado descargar la versión LTS)
- NPM: al instalar Node, se instala automáticamente.
- Un editor de código: podemos usar cualquier editor de texto, aunque es recomendable alguno que nos de soporte al lenguaje que utilicemos.
    - Visual Studio Code: link::https://code.visualstudio.com/[https://code.visualstudio.com/] (Recomendado)
    - Sublime Text: link::https://www.sublimetext.com/[https://www.sublimetext.com/]
    - Atom: link::https://atom.io/[https://atom.io/]
    - Brackets: link::http://brackets.io/[http://brackets.io/]
    - WebStorm: link::https://www.jetbrains.com/es-es/webstorm/[https://www.jetbrains.com/es-es/webstorm/] (De pago. Gratis para estudiantes)

Chapter 3. Instalación

Para poder instalar TypeScript necesitamos tener instalado NPM. Comprobamos que lo tenemos instalado lanzando el siguiente comando que nos tiene que mostrar la versión que tenemos:

```
$ npm -v
```

En caso de tenerlo instalado, entonces ejecutamos el siguiente comando para realizar la instalación de TypeScript:

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

Para asegurarnos de que se ha instalado correctamente, podemos lanzar el comando:

```
$ tsc -v
```

Chapter 4. Compilar a JavaScript

Al final cuando trabajamos con TypeScript, este código no se puede ejecutar directamente, ya que por ejemplo los navegadores no saben interpretarlo, por lo que tendremos que convertirlo primero a algo que si entiendan para ejecutarlo después.

El código de TypeScript lo tenemos que transpilar a JavaScript, el lenguaje que si que va a poder ejecutarse en el navegador, o incluso con Node.

Para poder realizar esta transpilación, vamos a utilizar el TSC que se ha instalado junto a TypeScript.

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.

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

```
$ tsc archivo.ts
```

El TSC incluye tambien el **modo watchers** que permite vigilar los cambios en los archivos de TypeScript, y cada vez que se realiza algún cambio en uno de ellos, este se encarga de compilar el archivo auntomáticamente sin que tengamos que intervenir lanzando de nuevo el comando anterior.

Para realizar la transpilación del código en este modo solo tenemos que añadir la opción **-w** como aparece a continuación:

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

4.1. Lab: Compilar a JavaScript

En este laboratorio vamos a ver como compilar un archivo con código de TypeScript a JavaScript.

Empezamos creando un archivo de TypeScript **main.ts** en el que vamos a poner el siguiente código.

/typescript-compilar-a-js-lab/main.ts

```
const msg: string = 'Hola mundo!';

console.log(msg)
```

Una vez que ya lo tenemos, vamos a proceder a transpilarlo para obtener el archivo de JavaScript que ejecutaremos después.

Primero lanzamos el siguiente comando desde la carpeta **typescript-compilar-a-js-lab** para transpilar el código:

```
$ tsc main.ts
```

Ahora ha debido de crear un archivo **main.js** al mismo nivel con el código equivalente en JavaScript.

/typescript-compilar-a-js-lab/main.js

```
var msg = 'Hola mundo!';
console.log(msg);
```

Si nos fijamos, aquellas características propias de TypeScript como el tipo que habíamos puesto a la constante, han desaparecido en el archivo generado.

Por último, como ya tenemos el código en JavaScript, ya podríamos ejecutarlo por ejemplo con Node, lanzando el siguiente comando:

```
$ node main.js
```

Chapter 5. Tipos de datos

El sistema de tipado que nos proporciona TypeScript nos permite definir y validar los tipos de los valores asignados a una variable. Esto se asegura de que el código se comporta como se espera que lo haga evitando que las variables cambien de tipo sin darnos cuenta como puede ocurrir en JavaScript.

Los tipos que podemos encontrarnos son:

Tipo de dato	**Keyword**	**Descripción**
Número	number	Se usa para representar tanto números enteros como fracciones.
String	string	Representa una cadena de caracteres.
Booleano	boolean	Se usa para representar valores lógicos (true o false).
Void	void	Se usa para indicar que una función no devuelve ningún tipo de valor.
Cualquiera	any	Es un tipo dinámico, es decir, que acepta cualquier tipo de los mencionados antes. Se le puede cambiar el valor independientemente del tipo.

Hay otros tipos de datos que veremos más adelante como los **arrays** y los **objetos**.

Chapter 6. Variables

En TypeScript podemos declarar las variables e inicializarlas, y según la forma en que lo hagamos dicha variable podrá guardar datos de unos tipos o de otros.

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';
```

Al declarar la variable no siempre la vamos a inicializar, pero lo podemos asignar el tipo de datos que puede guardar para evitar efectos indeseados en la aplicación.

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

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;
```

Por último podemos declarar una variable sin darle un valor ni indicarle de que tipo va a ser. En este caso la variable tendra 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;
```

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;
```

Chapter 7. Operadores

Los operadores realizan cambios sobre los datos y los podemos clasificar en distintos grupos.

7.1. 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)	a=1; a++; ⇒ a=2;
— (Decremento)	a=5; a--; ⇒ a=4;

7.2. Operadores relacionales

Operador	Ejemplo
< (Menor que)	3 < 7 (true)
> (Mayor que)	8 > 9 (false)
<= (Menor o igual que)	2 <= 2 (true)
>= (Mayor o igual que)	9 >= 4 (true)
== (Igual a)	3 == 2 (false)
!= (Distinto de)	4 != 5 (true)

7.3. Operadores lógicos

Operador	Ejemplo
&& (AND)	10 > 3 && false (false)
|| (OR)	10 > 3 || false (true)
! (NOT)	!(10 > 3) (false)

7.4. Operadores de asignación

Suponiendo que:

- a: 5
- b: 4

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

7.5. Typeof

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

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

7.6. Operador ternario

Este operador es equivalente al if...else.

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

7.7. Operador de Coalescencia Nula

El operador de coalescencia nula es como una versión más simple del operador ternario que hemos visto anteriormente, pero con alguna pequeña variación al obtener el resultado final.

Esta vez se utiliza **??** como operador.

```
// [variable o valor] ?? [variable o valor]
null ?? 1 // 1
```

Ummm Pero esto ya lo podíamos hacer:

```
(null ? null : 1) === (null ?? 1)
```

Incluso, podíamos hacerlo de esta otra forma:

```
(null ?? 1) === (null || 1)
```

Exacto, estos ejemplos son equivalentes, pero ¿qué pasa si cambiamos el **null** por un **0**?.

```
(0 ? 0 : 1) === (0 ?? 1)
(0 ?? 1) === (0 || 1)
```

Exacto, ya no son equivalentes. Si lo que hay a la izquierda tiene valor (distinto de null o undefined) se devuelve esta parte, pero si no es asi, entonces se devuelve lo que haya a la derecha.

Este nuevo operador nos vendrá bien cuando no querramos que se trate como falsy los valores que normalmente lo son como un 0 o un string vacío.

Chapter 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 algun elemento hay que poner todos los tipos separados por `|`.

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

Chapter 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;
```

Chapter 10. Template literals

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 concatenandolos junto al texto que queremos mostrar, y esto puede darnos bastante trabajo si tenemos que mostrar bastantes valores almacenados en variables.

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}`;
```

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`;
```

Chapter 11. 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.

11.1. Tuplas etiquetadas

En la versión 4 de TypeScript se han añadido las **tuplas etiquetadas**. Este nuevo tipo de tuplas no cambia el funcionamiento, sino que hace que sean mucho más legibles al desarrollador.

Con este tipo de tuplas, vamos a poder indicar que tipo de dato se va a guardar en cada posición de la tupla.

Para ello, solo tenemos que poner el nombre del dato seguido del tipo de dato.

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

Chapter 12. Desestructuración

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

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;
```

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;
```

Chapter 13. Estructuras condicionales

Las estructuras condicionales se encargan de ejecutar un bloque de código según la condición que se cumpla.

13.1. 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');
}
```

13.2. Ejercicio

- Crear el juego de Fizz-Buzz
  - Consiste en preguntar un número al usuario
  - Si el número es múltiplo de 3 se muestra el mensaje *Fizz* junto al número
  - Si el número es múltiplo de 5 se muestra el mensaje *Buzz* junto al número

13.3. 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');
}
```

13.4. Ejercicio

- Preguntar al usuario si quiere que le contemos un chiste
  - Si la respuesta es *si* le mostramos un chiste en la consola
  - Si la respuesta es *no* no le mostramos un mensaje al aburrido

13.5. 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');
}
```

13.6. SWITCH

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 coincida 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 coincida con el valor que se está comparando.

Al final de cada `case`, se pone la palabra instrucción `break` que le dice al interprete de JavaSript que ya ha terminado de ejecutar el `switch` y que puede seguir ejecutando el código a partir de su bloque.

```
switch (nota) {
  case 5:
    console.log('Suficiente');
    break;
  case 6:
    console.log('Bien');
    break;
  case 7:
    console.log('Bien alto');
    break;
  case 8:
    console.log('Notable');
    break;
  case 9:
      console.log('Notable alto');
      break;
  case 10:
    console.log('Sobresaliente');
```

```
    break;
  default:
    console.log('Suspenso');
    break;
}
```

13.7. 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));
}
```

13.8. Ejercicio

- Haz un programa que cuente hacia atrás 30 números.

13.9. Ejercicio

- Mejora el ejercicio de Fizz Buzz de antes para que muestre todos los números hasta llegar al que se ha introducido.

13.10. 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 (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);
}
```

13.11. FOR OF

Este bucle 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);
}
```

13.12. Ejercicio

- Recorrer el siguiente array y el siguiente objeto, mostrando todos los datos de cada uno de ellos.

```
let colores = ['Blanco', 'Negro', 'Amarillo', 'Azul', 'Verde', 'Naranja', 'Rojo'];

let superheroe = {
  nombre: 'Wade',
  apellido: 'Wilson',
  alias: 'Deadpool'
};
```

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

13.14. Ejercicio

- Crea un programa que muestre los 10 primeros números que son múltiplos de 7.

13.15. Ejercicio

- Mostrar en la consola la canción de Homer Simpson, cojo un muelle y lo tiro por el retrete... hasta llegar a tantos muelles como haya escrito el usuario en un popup.

13.16. 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);
```

13.17. Ejercicio

- Adivina el número al azar
  - El programa saca un número al azar entre 0 y 50
  - El usuario introduce el número que piensa que ha salido
  - Si el usuario acierta se le indica que ha ganado, el número de turnos jugados, y se le pregunta si quiere volver a jugar
  - Si el usuario no acierta:
    - Se le indica si el número que ha dicho es mayor o menor
    - Se le vuelve a pedir un número

Chapter 14. Funciones

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 quieres ejecutar el bloque de código que contiene. A las funciones se les pueden pasar parámetros y algunas devuelven valores.

Las diferencias que hay entre las funciones de TypeScript y las de JavaScript son las siguientes:

- **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.
  - Soporta **arrow functions**.
  - Los parámetros pueden ser requeridos u opcionales.
  - Se le pueden poner valores por defecto a los parámetros.
  - Soporta parámetros **rest**.
  - Permite sobrecargar funciones.
- **JavaScript**
  - No tiene tipos.
  - Soporta **arrow functions** en ES6.
  - Todos los parámetros son opcionales.
  - Se le pueden poner valores por defecto a los parámetros en ES6.
  - No se permite sobrecargar funciones.
  - Soporta parámetros **rest** en ES6.

14.1. 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;
}
```

14.1.1. Ejercicio

- Crear una función que compruebe si el segundo parámetro puede ser el valor de la sigla que recibe como primer parámetro.
  - 'E.S.O' ⇒ 'Educación Secundaria Obligatoria'

14.2. Arrow functions

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.

Hay muchos métodos que reciben como parámetro una función anónima, y es muy posible que esta función anónima tenga solo una linea de código. Para esta sola linea de código estariamos usando la palabra reservada `function`, habria que abrir llaves `{}` y en caso de devolver un valor usar la palabra reservada `return`. Las arrow functions nos ahorran el tener que escribir todo ese código para solo una linea que va a contener en el cuerpo, y se llaman arrow functions porque el cuerpo y los parámetros van separados por una flecha (`⇒`).

```
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');
```

Cuando estas funciones no reciben parámetros o reciben más de uno, entonces hay que ponerlos entre parentesis, mientras que si reciben solo uno, da igual si ponemos los parentesis 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 + ' (' +
pelicula.genero + ')' + ' vista!'));
misPeliculas.forEach((pelicula, index) => {
  let texto = pelicula.titulo + ' (' + pelicula.genero + ')' + ' vista!';
  console.log(texto);
});
```

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);
```

```
  });
}

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

14.3. Definición de tipos de funciones

Al igual que a una variable le podemos indicar de que tipo va a ser el valor que se le va a asignar, con las funciones pordemos hacer lo mismo diciendole la forma que tiene que tener la 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);
```

14.4. Parámetros opcionales y por defecto

Podemos indicarle a una función que un parámetro es opcional poniendole el símbolo de interrogación despues del nombre del parámetro. Todos los parámetros opcionales deben de ir situados despues de todos aquellos que son obligatorios. Y para asignarle un valor por defecto a un parámetro le igualamos el valor a ese parámetro en la sección donde se definen los parámetros.

```
function muestraDatosUsuario(nombre: string = 'Blanca', email?: string): void {
  let datos = nombre;
  if (email != undefined) {
    datos += ' - ' + email;
  }
  console.log(datos);
}

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

```
muestraDatosUsuario();
```

14.4.1. Ejercicio

- Crear una función que dado un array de strings, tiene que devolver un string con todos los elementos del array concatenados.
  - Se puede pasar un string como parámetro para ponerlo al final del string que se va a devolver.
  - Tiene que recibir un separador como argumento para añadirlo entre cada palabra.

14.5. 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);
```

14.5.1. Ejercicio

- Crear una función que nos devuelva la media de una lista de números.

14.6. 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. Para poder hacer esto, tendremos que definir la función con sus distintos parámetros y luego implementar la función en la que dependiendo de los parámetros que nos lleguen, hará una cosa u otra.

```
function doble(valor: number);
function doble(valor: string);

function doble(valor: any) {
  const tipo = typeof valor;
  if (tipo == 'number') {
    return valor * 2;
  } else {
    return valor + valor;
  }
}

let dobleDe4 = doble(4);
```

```
let dobleDeHola = doble('Hola');
```

14.6.1. Ejercicio

- Sobrecargar una función para devolvernos un array de peliculas que cumplen con un filtro.
- Hay que poder filtrar por:
  - Nombre
  - Genero
  - Año
  - Estrenada

Chapter 15. Interfaces

Las interfaces son como contratos que definen como va a ser un tipo de datos. Estas interfaces no se compilan a nada en JavaScript. Al transpilar nuestro código TypeScript, se mostrarán warnings en caso de que no se esté cumpliendo lo que se ha definido en la interfaz. 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.

15.1. Definir una interfaz

Una interfaz se define de la siguiente forma:

```
interface Pelicula {
  id: number,
  titulo: string,
  genero: string,
  duracion?: number, // Parámetro opcional
  ganadoraOscar?: (gana: boolean) => void // Los métodos sin implementar
}

let pelicula: Pelicula = {
  id: 12,
  titulo: 'Los mercenarios',
  genero: 'accion',
  duracion: 113,
  ganadoraOscar: (gana: boolean) => console.log(gana ? 'Ha ganado un oscar' : 'No ha
ganado ningún oscar')
}

pelicula.ganadoraOscar(true);
pelicula.ganadoraOscar(false);
```

15.1.1. Ejercicio

- Definir una interfaz para crear objetos **Coche** que tengan las siguientes propiedades y metodos
  - Marca
  - Matricula
  - Color
  - Sonido
  - TocarClaxon (metodo)
  - Pintar (metodo)

15.2. Interfaces para tipos de función

En el tema de las funciones hemos visto como podriamos crear un tipo para las funciones, pero habia que repetir bastante código. Ahora vamos a ver como podemos darles nombre a los tipos de funciones usando interfaces, que nos ayudarán a reutilizar código y hacerlo más legible.

```
interface generadorStrings {
  (chars: string, nums: number): string
}

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

let generadorIds: generadorStrings = crearUsuarioId;
let generadorIds2: generadorStrings = crearUsuarioId;

let miId1 = generadorIds('angel', 201);
let miId2 = generadorIds('pedro', 104);
```

15.3. Extendiendo interfaces

Las interfaces pueden ser extendidas y componer otras interfaces, de está forma podemos crear tipos que usan otros tipos. Estas interfaces que extienden otras interfaces tienen que tener las propiedades y métodos de las interfaces que extienden. Para extender interfaces se usa la palabra reservada `extends` seguida de las interfaces que va a extender.

```
interface Persona {
  nombre: string,
  email: string
}

interface DirectorCine extends Persona {
  numPeliculasDirigidas: number
}

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

15.4. Tipos de clases

Como hemos dicho antes, las interfaces no se encargan de crear objetos o de implementar los métodos definidos en ellas, sino que eso es tarea de las clases. Para indicar que una clase tiene que

hacer uso de lo que se ha definido en una interfaz, tiene que usar la palabra reservada `implements` seguida de las interfaces que tiene que implementar.

```
interface Desarrollador {
  trabaja: () => void;
}

class DesarrolladorJavascript implements Desarrollador {
  trabaja() {
    console.log('Desarrollo aplicaciones con JavaScript');
  }
}

let desarrollador: Desarrollador = new DesarrolladorJavascript();
desarrollador.trabaja();
```

15.5. Fusión de declaraciones

La **fusión de declaraciones** consiste en que el compilador va a mergear dos declaraciones separadas, que se han declarado con el mismo nombre, en una sola definición. No es posible mergear clases con otras clases.

app.ts

```
interface Libro {
  nombre: string;
  autor: string;
  año: number;
  categoria: string;
}

interface Libro {
  isbn: string;
  pais: string;
  editorial: string;
}

let libro: Libro = {
  nombre: 'Donde surgen las sombras',
  autor: 'David Lozano Garbala',
  año: 2006,
  categoria: 'Suspense',
  isbn: '9788467510270',
  pais: 'España',
  editorial: 'SM'
};

console.log(libro);
```

Chapter 16. Clases

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 y que se relaciona con una entidad (cualquier cosa).

16.1. Constructores

Los constructores se encargan de crear nuevas instancias de una clase. Los constructores son métodos cuyo nombre es `constructor` y solo puede haber uno por clase. Los constructores pueden recibir parámetros que van a servir para darle valores a las propiedades de los objetos que vamos a crear. Para crear objetos se usa la palabra `new` seguida del nombre de la clase con los parámetros que espera recibir entre paréntesis que se encargará de ejecutar la función constructora.

```
class Mascota {
  constructor(nombre: string, tipo: string, sonido: string) { ... }
}

let perro = new Mascota('Rocky', 'perro', 'guau');
```

16.2. Propiedades y métodos

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;
  public sonido: string;

  constructor(nombre: string, tipo: string, sonido: string) {
    this.nombre = nombre;
    this.tipo = tipo;
    if (sonido) {
      this.sonido = sonido;
    }
  }

  toString (): void {
    console.log(`Mi ${this.tipo} ${this.nombre} hace ${this.sonido}`);
  }
}

let perro = new Mascota('Rocky', 'perro', 'guau');
perro.toString();
```

Hay dos formas de inicializar las propiedades, una es declarando la propiedad y en el constructor le asignamos el valor que llega como parámetro, como en el ejemplo anterior.

Y la otra es una forma abreviada de hacer lo anterior. Hay que indicar la visibilidad de la propiedad en el propio constructor justo antes del nombre de la variable. De esta forma está declarando la propiedad e inicializandola al mismo tiempo.

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

  toString (): void {
    console.log(`Mi ${this.tipo} ${this.nombre} hace ${this.sonido}`);
  }
}

let perro = new Mascota('Rocky', 'perro', 'guau');
perro.toString();
```

También podemos tener propiedades estáticas, que no son propiedades de la instancia, sino que son propiedades de la clase. Lo que significa que 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;
```

16.2.1. Ejercicio

- Crear una clase que represente una Serie. Esta tiene que tener las siguientes propiedades y métodos:
  - Titulo
  - Año de estreno
  - Episodios
  - Temporadas
  - Episodios Vistos
  - Finalizada/Cancelada
  - Ver episodio (metodo)
  - Episodios por ver (metodo)
  - Serie Vista (metodo)

- toString (metodo)

16.3. Herencia

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` seguida de la clase de la cual está heredando la funcionalidad. 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.

```
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}
    `);
  }
}

let alumno = new Alumno('Benito', '834289342E', 7367343, 16);
alumno.toString();
```

16.4. Visibilidad (Public, Private and Protected)

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).

Si queremos indicar de forma explicita 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}`);
  }
}
```

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}`);
  }
}
```

Y por último, aunque en JavaScript no exista, 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 actua 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}`);
  }
}
```

```
class SubPersonaProtegida extends PersonaProtegida {
  constructor(nombre: string) {
    super(nombre);
  }
  toString() {
    // Aquí podemos acceder al nombre
    console.log(`Me llamo ${this.nombre}`);
  }
}
```

16.5. 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}`);
  }
}
```

16.6. 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.

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}`);
  }
}
```

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) { }
  get nombre (): string {
    return this._nombre;
  }
  set nombre(nombre: string) {
    this._nombre = nombre;
  }
```

```
  toString() {
    console.log(`Me llamo ${this.nombre}`);
  }
}
```

Como buena practica, 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 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
```

16.7. Clases Abstractas

Las **clases abstractas** son clases que van a ser heredadas por otras clases. En la clase abstracta se implementará todos los métodos que van a compartir las clases que heredan de esta, y así se evita repetir el código en las distintas clases. 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);
  }
}

class Perro extends Animal {
  constructor(nombre: string) {
    super(nombre);
  }
}

let perro = new Perro('Balto');
console.log(perro.nombre);
```

```
let gato = new Gato('Felix');
gato.nombre = 'Gardfield';
console.log(gato.nombre);
```

En las clases abstractas se pueden declarar **métodos abstractos** que no van a ser implementados en dicha clase, sino que los implementarán aquellas clases que heredan de la abstracta. Estos métodos llevan la palabra `abstract` delante del nombre y solo hay que declararlo indicando los parámetros y el tipo de datos que devuelve.

```
abstract class Animal {
  constructor(protected _nombre: string) { }

  get nombre() {
    return this._nombre;
  }

  set nombre(nombre: string) {
    this._nombre = nombre;
  }

  abstract hazSonido(): void;
}
```

```
class Gato extends Animal {
  constructor(nombre: string) {
    super(nombre);
  }

  hazSonido(): void {
    console.log('Miauuu');
  }
}

class Perro extends Animal {
  constructor(nombre: string) {
    super(nombre);
  }

  hazSonido(): void {
    console.log('Guauu Guauu');
  }
}

let perro = new Perro('Balto');
perro.hazSonido();
let gato = new Gato('Felix');
gato.hazSonido();
```

16.7.1. Ejercicio

- Crear un validador de movimientos de Ajedrez
  - Crear una clase Pieza
  - Crear las piezas *Torre*, *Rey* y *Peón*
  - Comprobar que las piezas se pueden mover a una posicion dada (es un movimiento permitido para la pieza)

Chapter 17. Enumerados

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 númerico que empieza en 0.

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

/enums.ts

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

Podemos establecer estos valores nosotros mismos igualandolos 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.

/enums.ts

```
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]);
```

Chapter 18. Genéricos

Los genéricos son bloques de código que nos permiten trabajar con múltiples tipos de datos. Los genéricos aceptan tipos como parámetros. 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 que tipo de datos se van a realizar operaciones.

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

18.1. Funciones genéricas

Podemos usar los genéricos con las funciones. La función recibirá como parámetro el tipo T justo antes de la lista de parámetros.

```
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']);
```

18.2. Interfaces y clases genéricas

También es posible de crear interfaces y clases con tipos genéricos.

```
interface Inventario<T> {
  addItem: (item: T) => void;
  getItemsInventario: () => Array<T>;
}

interface Portatil {
  marca: string;
}

class Catalogo<T> implements Inventario<T> {
  private catalogo = new Array<T>();

  addItem(item: T) {
    this.catalogo.push(item);
  }

  getItemsInventario(): Array<T> {
```

```
    return this.catalogo;
  }
}

let catalogoPortatil = new Catalogo<Portatil>();
catalogoPortatil.addItem({marca: 'HP'});
catalogoPortatil.addItem({marca: 'Lenovo'});
let items = catalogoPortatil.getItemsInventario();
```

18.2.1. Ejercicio

- Crear una clase que funcione como un Map generico (objeto clave/valor).
- Tiene que tener las siguientes funcionalidades:
  - Añadir un nuevo elemento
  - Obtener un elemento
  - Comprobar si existe un elemento
  - Vaciar la lista
  - Mostrar los items que hay en el Map

Chapter 19. Módulos

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`. Hay dos formas de hacerlo.

mascota.ts

```
export class Mascota {
  constructor(public nombre: string, public tipo: string, public sonido: string) { }
  hazSonido(): void {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
}
```

mascota.ts

```
class Mascota {
  constructor(public nombre: string, public tipo: string, public sonido: string) { }
  hazSonido(): void {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
}
export { Mascota };
```

Y 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.

app.ts

```
import { Mascota as Masc } from './mascota';

let perro = new Masc('Toby', 'perro', 'guau');
perro.hazSonido();
```

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.

app.ts

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

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

19.1. Exportando por defecto

Hay otra opción para exportar a parte de las vistas antes, 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.

mascota.ts

```
export default class {
  constructor(public nombre: string, public tipo: string, public sonido: string) { }
  hazSonido(): void {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
}
```

app.ts

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

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

Chapter 20. Namespaces

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.

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.

app.ts

```
namespace Validator {
  const letras = /^\w[^\d_.]+$/g;
  const numeros = /^\d+$/g;

  export function soloLetras(cadena: any) {
    return letras.test(cadena);
  }
  export function minusculasYMayusculas(cadena: any) {
    return (cadena.toLowerCase() !== cadena) && (cadena.toUpperCase() !== cadena);
  }
  export function soloNumeros(cadena: any) {
    return numeros.test(cadena);
  }
  export function noVacio(cadena: any) {
    return cadena.length > 0;
  }
  export function longitudMinima3(cadena: any) {
    return cadena.length >= 3;
  }
}

console.log(Validator.noVacio(''));
console.log(Validator.soloLetras('h0la'));
console.log(Validator.soloLetras('hola'));
console.log(Validator.minusculasYMayusculas('hola'));
console.log(Validator.longitudMinima3('Ey'));
console.log(Validator.soloNumeros(123));
```

Una vez que tenemos el namespace creado, compilamos el archivo y comprobamos que salen los valores esperados.

Hay veces que los namespaces se pueden volver demasiado grandes y dificiles de mantener, por lo que es una buena idea separarlos en distintos archivos.

app.ts

```
console.log(Validator.noVacio(''));
console.log(Validator.soloLetras('h0la'));
console.log(Validator.soloLetras('hola'));
console.log(Validator.minusculasYMayusculas('hola'));
console.log(Validator.longitudMinima3('Ey'));
console.log(Validator.soloNumeros(123));
```

validar-letras.ts

```
namespace Validator {
  const letras = /^\w[^\d_.]+$/g;

  export function soloLetras(cadena: any) {
    return letras.test(cadena);
  }
  export function minusculasYMayusculas(cadena: any) {
    return (cadena.toLowerCase() !== cadena) && (cadena.toUpperCase() !== cadena);
  }
}
```

validar-numeros.ts

```
namespace Validator {
  const numeros = /^\d+$/g;

  export function soloNumeros(cadena: any) {
    return numeros.test(cadena);
  }
}
```

validar-longitud.ts

```
namespace Validator {
  export function noVacio(cadena: any) {
    return cadena.length > 0;
  }
  export function longitudMinima3(cadena: any) {
    return cadena.length >= 3;
  }
}
```

Pero de esta forma en el archivo donde se están usando las validaciones no se tiene acceso a los namespaces, por lo que tendremos que importarlos de alguna forma.

Esto lo podemos hacer de dos formas distintas.

La primera es compilar todos los archivos e importarlos en un archivo `index.html` antes del `app.js`.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <script src="validar-letras.js"></script>
  <script src="validar-numeros.js"></script>
  <script src="validar-longitud.js"></script>
  <script src="app.js"></script>
</head>
<body>
</body>
</html>
```

SI abrimos la página deberían de salir los resultados esperados.

La otra forma de hacerlo es importando los namespaces dentro del archivo principal con la siguiente etiqueta.

```
/// <reference path="archivo.ts"/>
```

Esta etiqueta indica que esos archivos se tienen que incluir en la compilación, y de esta forma se podrá acceder al namespace desde el archivo principal.

app.ts

```
/// <reference path="validar-letras.ts"/>
/// <reference path="validar-numeros.ts"/>
/// <reference path="validar-longitud.ts"/>

console.log(Validator.noVacio(''));
console.log(Validator.soloLetras('h0la'));
console.log(Validator.soloLetras('hola'));
console.log(Validator.minusculasYMayusculas('hola'));
console.log(Validator.longitudMinima3('Ey'));
console.log(Validator.soloNumeros(123));
```

Si probamos a ejecutar este archivo, nos va a dar un error. Esto es porque el código perteneciente a los distintos namespaces no se ha incluido en el archivo `app.js` que resulta de la compilación.

Para añadirlos vamos a usar la opción **outFile** en la configuración del compilador. Y esta opción recibe como valor el nombre del archivo que se tiene que generar con todo el código JavaScript correspondiente.

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "outFile": "app.js"
  }
}
```

Si ahora se prueba a ejecutar ya tiene que funcionar correctamente.

20.1. Ejercicio

- Crear un namespace para mostrar una palabra/frase en español traducida a inglés, francés e italiano.
  - Separar el namespace en distintos archivos (uno por idioma).
  - Guardar las traducciones en un objeto dentro del namespace.

Chapter 21. Mixins

Los **mixins** son un tipo de intersección, son clases cuya funcionalidad se le agrega a otras clases.

En el siguiente código vemos un mixin, que se podrá usar con distintos objetos para indicar si están disponibles o no.

mixins.ts

```
class Disponible {
  estaDisponible: boolean;
  ponerDisponible() {
    this.estaDisponible = true;
  }
}

export { Disponible };
```

Y ahora vamos a usar ese mixin en dos clases, una es *Persona* y la otra *HabitaciónHotel*. De está forma podremos saber si una persona está disponible por ejemplo para una reunión y en el caso de la habitación del hotel para saber si esta está disponible para reservarla. El mixin le da la misma funcionalidad a ambas clases sin que estas estén relacionadas entre sí. Estas clases implementan los mixins, y tienen que tener dentro de la clase los mismos nombres de propiedades y métodos que el mixin, para que la función `applyMixins` funcione correctamente.

app.ts

```
import { Disponible } from './mixins';

class Persona implements Disponible {
  nombre: string;

  constructor(nombre: string) {
    this.nombre = nombre;
  }

  informacion(): void {
    console.log('Soy ' + this.nombre + ' y ' + (this.estaDisponible ? '' : 'no ') +
'estoy disponible para la reunión.');
  }

  // Disponible
  estaDisponible: boolean = false;
  ponerDisponible: () => void;
}

class HabitacionHotel implements Disponible {
  nombreHotel: string;
  numeroHabitacion: number;
```

```
  constructor(nombreHotel: string, numeroHabitacion: number) {
    this.nombreHotel = nombreHotel;
    this.numeroHabitacion = numeroHabitacion;
  }

  informacion(): void {
    console.log('La habitación número ' + this.numeroHabitacion + ' del hotel ' +
this.nombreHotel + (this.estaDisponible ? '' : ' no') + ' esta disponible para ese
día.');
  }

  // Disponible
  estaDisponible: boolean = false;
  ponerDisponible: () => void;
}


function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach(baseCtor => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
      derivedCtor.prototype[name] = baseCtor.prototype[name];
    });
  });
}

applyMixins(Persona, [Disponible]);
applyMixins(HabitacionHotel, [Disponible]);

let persona = new Persona('Robb');
persona.informacion();
let habitacion = new HabitacionHotel('Hilton', 123);
habitacion.informacion();

persona.informacion();
persona.ponerDisponible();
persona.informacion();

habitacion.informacion();
habitacion.ponerDisponible();
habitacion.informacion();
```

La función `applyMixins(derivedCtor: any, baseCtors: any[])` se encarga de añadir la funcionalidad de los mixins a una clase. El primer parámetro es la clase a la que se le quiere añadir los mixins, y el segundo parámetro es un array de mixins a añadir.

Chapter 22. Promesas

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**).

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**).

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.

app.ts

```
let promesa: Promise<Array<number>> = new Promise<Array<number>>(getNumeros);

function getNumeros (resolve, reject) {
  let xhr = new XMLHttpRequest();
  xhr.open('GET', 'datos.json');
  xhr.onreadystatechange = function () {
    if (xhr.status === 200 && xhr.readyState === 4) {
      let nums = JSON.parse(xhr.responseText).numeros;
      if (nums.length > 0) {
        resolve(nums);
      } else {
        reject(new Error('No hay numeros'));
      }
    }
  }
  xhr.send();
}
```

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*.

app.ts

```
let promesa: Promise<Array<number>> = new Promise<Array<number>>((resolve, reject) =>
{
  let xhr = new XMLHttpRequest();
  xhr.open('GET', 'datos.json');
  xhr.onreadystatechange = function () {
    if (xhr.status === 200 && xhr.readyState === 4) {
      let nums = JSON.parse(xhr.responseText).numeros;
      if (nums.length > 0) {
        resolve(nums);
      } else {
        reject(new Error('No hay numeros'));
      }
    }
  }
  xhr.send();
});
```

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).

app.ts

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

22.1. 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.

app.ts

```
const nums: Array<number> = [1, 2, 3, 4, 5, 9, 9, 7, 3, 5];
function getDobleDeNumerosPares() {
  return new Promise<Array<number>>((resolve, reject) => {
    setTimeout(() => {
      if (nums.length > 0) {
        resolve(nums);
      } else {
        reject('El array está vacío');
      }
    }, 2000);
```

```
  });
}

getDobleDeNumerosPares().then((numeros) => {
  console.log('Números: ' + numeros);
  let pares = numeros.filter(n => n % 2 == 0);
  if (pares.length == 0) throw new Error('No hay números pares');
  return pares;
}).then(numsPares => {
  console.log('Números pares: ' + numsPares);
  return numsPares.map(n => n*2);
}).then(numsDobles => {
  console.log('Numeros * 2: ' + numsDobles);
}).catch((error) => {
  console.log('Error: ' + error.message);
});
```

22.1.1. Ejercicio

- Crear una promesa que pida unos generos de pelicula a una BBDD:
  - Tiene que mostrar los generos disponibles en la pantalla y que te permita seleccionar uno de ellos (usar prompt).
  - Tiene que pedir las peliculas del genero elegido a la BBDD para mostrarlas.
  - Hay que encadenar las promesas

```
{
  'generos': ['action', 'terror'],
  'peliculas': {
    'action': ['Peli 1', 'Peli 2, 'Peli 3'],
    'terror': ['Peli 4']
  }
}
```

Chapter 23. Compilador de Typescript

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`.

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

```
$ tsc --init
```

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

tsconfig.json

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

En este fichero nos vamos a encontrar con un objeto en el que podemos encontrar las siguientes opciones:

Table 1. TSConfig Options

Opción	Descripción
compilerOptions	Opciones a tener en cuenta cuando se vaya a compilar los archivos de typescript.
files	Es un array de archivos que se van a compilar, aquellos que no se añaden aquí no se transpilan a javascript (usando rutas absolutas y relativas).
include	Es un array donde se indican los ficheros que queremos que se compilen (usando expresiones regulares).
exclude	Es un array donde se indican los ficheros que no queremos que se compilen (usando expresiones regulares).

tsconfig.ts

```
{
```

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

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 la página oficial. Vamos a ver algunas de ellas.

Empezamos por indicar que queremos generar los archivos de source map para poder debuggear la aplicación y para ello solo hay que cambiarle el valor a la opción **sourceMap**.

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

Vamos a generar un archivo `index.html` para servirlo y en el importaremos nuestra aplicación.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <script src="app.js"></script>
</head>
<body>
  <h1>Typescript</h1>
</body>
```

```
</html>
```

El código que se va a importar en el `index.html` es el siguiente.

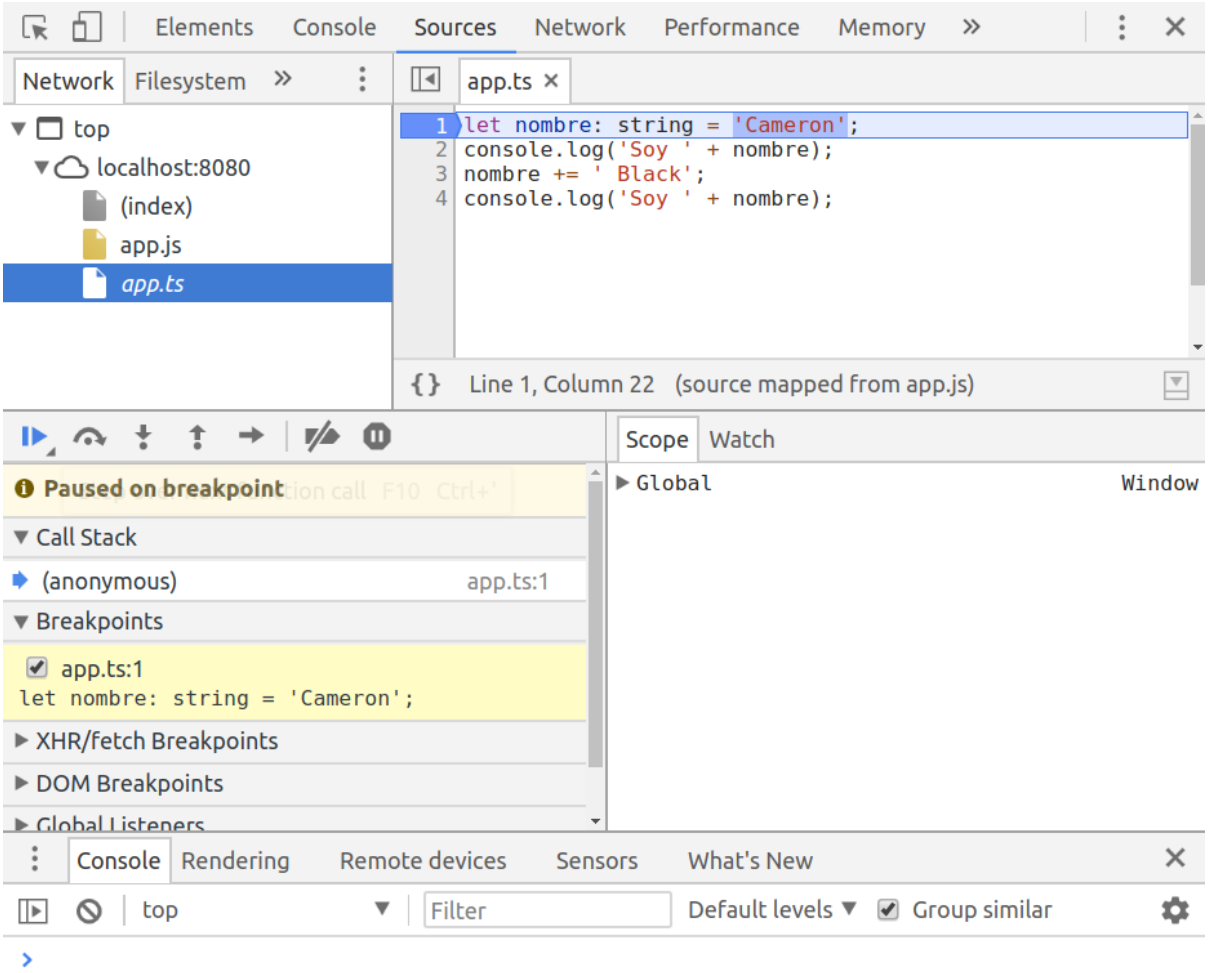
app.ts

```
let nombre: string = 'Cameron';
console.log('Soy ' + nombre);
nombre += ' Black';
console.log('Soy ' + nombre);
```

Ahora al compilar el archivo, se va a generar un archivo con la extensión `.js.map`. Ahora vamos a levantar la aplicación con un servidor local. Podemos usar el siguiente comando:

```
$ http-server
```

Ahora en el navegador en la pestaña de **source** se encuentra el archivo JavaScript que se ha generado y el archivo Typescript que hemos creado nosotros. En este archivo podemos poner puntos de interrupción y de esta forma podemos debuggear el código.



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 **noImplicitAny** que nos dará errores cuando esto ocurra.

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": true,
    "sourceMap": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

app.ts

```
function sumaNumeros(n1, n2) {
  return n1 + n2;
}

let suma1 = sumaNumeros(1, 2);

let suma2 = sumaNumeros(2, true);
```

En caso de que los argumentos no tengan un tipo específico, sino que pueden recibir cualquier dato, entonces vamos a evitar que nos muestre el error añadiendole explicitamente el tipo `any`.

app.ts

```
function mostrarDato(dato: any) {
  console.log(dato);
}
```

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**.

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
```

```
    "noImplicitAny": true,
    "sourceMap": true,
    "noEmitOnError": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

app.ts

```
let num: number = 4;
num = '4';
```

Y si ahora compilamos los archivos, no tienen que aparecer los `.js`.

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**.

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

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": true,
    "sourceMap": true,
    "noEmitOnError": true,
    "pretty": true,
    "removeComments": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

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

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": true,
```

```
    "sourceMap": true,
    "noEmitOnError": true,
    "pretty": true,
    "removeComments": true,
    "outDir": "./js"
  },
  "exclude": [
    "node_modules"
  ]
}
```

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 compilaran otra vez.

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": true,
    "sourceMap": true,
    "noEmitOnError": true,
    "pretty": true,
    "removeComments": true,
    "outDir": "./js",
    "watch": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

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.

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": true,
    "sourceMap": true,
    "noEmitOnError": true,
    "pretty": true,
    "removeComments": true,
    "outDir": "./js",
```

```
    "watch": false,
    "strictNullChecks": true
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "app.ts"
  ]
}
```

app.ts

```
function esMayorQue3(num: number) {
  let res: boolean;
  if (num > 3) {
    res = true;
  }
  return res;
}
```

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**.

tsconfig.json

```
{
  "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
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "app.ts"
  ]
```

```
}
```

app.ts

```
function getMensaje2(str1: string, str2: string) {
  let mensaje: string;
  return 'Mensaje: ' + str1;
}
```

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 **noImplicitReturns** para que nos avise si eso ocurre.

tsconfig.json

```
{
  "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
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "app.ts"
  ]
}
```

app.ts

```
function esPar(num: number) {
  if (num % 2 == 0) {
    return true;
  }
}
```

Chapter 24. Librerías externas JS

Hay muchas librerías JavaScript muy útiles que seguramente que vayamos a querer usar en nuestro código Typescript.

Para poder reutilizar estas librerías sin necesidad de escribirlas con Typescript, se crean los **type definitions** que son archivos en los que se definen la forma de todos los métodos y propiedades (*interfaces*) que tienen estas librerías de JavaScript para que hagan de puente entre nuestro código de Typescript y el código JavaScript de las librerías.

Al ser interfaces, todo ese código solo se va a usar para evitar errores durante la compilación (variables y métodos no definidos), y una vez que se han compilado nuestros archivos ese código desaparecerá porque en JavaScript no hay nada equivalente a las interfaces.

24.1. En el cliente

Primero vamos a usar una librería que depende del navegador como puede ser **SweetAlert**.

Lo primero de todo vamos a inicializar un proyecto de npm donde vamos a guardar las dependencias necesarias.

```
$ npm init
```

Una vez que se ha creado el archivo `package.json` vamos a instalar las dependencias necesarias.

```
$ npm install --save sweetalert
```

Una vez instalada el archivo queda de la siguiente forma:

package.json

```
{
  "name": "ej-sweetalert",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "sweetalert": "^2.1.0"
  }
}
```

Lo siguiente es crear el archivo html donde se usará la librería de SweetAlert para mostrar

mensajes de alerta. Y en este archivo incluiremos los scripts, tanto de la librería, como el nuestro para que se pueda usar esta librería.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
</body>
<script src="node_modules/sweetalert/dist/sweetalert.min.js"></script>
<script src="app.js"></script>
</html>
```

Y por último, en el archivo con nuestro código será donde vamos a lanzar un popup usando la librería instalada.

app.ts

```
swal('Hola mundo!!!');
```

Ahora vamos a compilar el archivo con el código de Typescript para generar el archivo de JavaScript y que al abrir el navegador se muestre el mensaje.

```
$ tsc app.ts
```

Si abrimos la página HTML, veremos que está funcionando correctamente y aparece un popup como el siguiente.

Pero si miramos en la consola justo después de compilar nuestro archivo, ahí aparece un error.

```
app.ts(3,1): error TS2304: Cannot find name 'swal'.
```

Este error se muestra porque Typescript no sabe lo que es `swal`. Funciona porque Javascript si que conoce esa función ya que el script de la librería se ha importado antes del nuestro.

Para solucionar este error de compilación, solo tenemos que indicar que hay una variable `swal` definida en algún lugar de todo el código.

Para ello vamos a usar la palabra reservada `declare` para declarar una variable. Pero en este caso no se crea una variable, sino que se encarga de indicarle al compilador que ya existe una variable declarada con ese nombre.

app.ts

```
declare var swal;

swal('Hola mundo!!!');
```

Si ahora volvemos a compilar, podremos observar que ya no se muestra el error de antes.

24.2. En el servidor

En este caso vamos a ver el proceso para usar una librería que no necesita que se encuentre el navegador abierto para funcionar, como puede ser **Axios**.

Vamos a empezar por inicializar el proyecto con *npm*:

```
$ npm init
```

Y una vez que se ha creado el archivo `package.json` vamos a instalar las dependencias necesarias.

```
$ npm install --save axios
```

Una vez instalado el archivo queda de la siguiente forma:

package.json

```
{
  "name": "ej-axios",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
```

```
  "dependencies": {
    "axios": "^0.18.0"
  }
}
```

Ahora vamos a usar la librería en nuestro archivo de Typescript. Para ello solo hay que importarla usando la sintáxis vista en el tema de los módulos.

app.ts

```
import axios from 'axios';

const URL = 'tareas.json';

let getTareas = axios.get(URL);
getTareas.then(({data}) => {
  console.log('Lista de tareas');
  console.log('---------------------');
  for (let id in data) {
    console.log('=> ' + data[id].tarea);
  }
  console.log('---------------------');
}).catch((error) => {
  console.log(`Error: ${error.message}`);
});
```

Y ahora solo queda compilar el archivo de Typescript y ejecutar el archivo JavaScript resultante con *node*.

```
$ tsc app.ts

$ node app.js
```

24.3. Typescript con Webpack

Webpack es un empaquetador de módulos, es decir, permite generar un archivo único (`bundle.js`) con todos aquellos módulos necesarios para el funcionamiento de la aplicación.

Webpack nos soluciona algunos de los problemas que surgen al desarrollar aplicaciones web, y sobre todo cuando son muy grandes:

- Múltiples peticiones
- Tamaño del código
- Orden de los archivos de dependencias
- Transpilación
- Linting

Empezamos por crear un proyecto con NPM:

```
$ npm init
```

Una vez que tenemos el archivo `package.json` vamos a instalar las dependencias que vamos a necesitar.

- **webpack**: el empaquetador de módulos.
- **webpack-dev-server**: servidor de desarrollo local de Webpack.
- **webpack-cli**: linea de comandos de Webpack.
- **typescript**: Typescript.
- **ts-loader**: loader de Typescript. Se encarga de compilar el código de Typescript a JavaScript.

```
$ npm install --save webpack webpack-dev-server webpack-cli typescript ts-loader
```

Una vez instaladas las dependencias, vamos a modificar el archivo de configuración de Typescript. Ahora no necesitamos indicar que archivos tiene que excluir o incluir en la compilación, porque de eso se va a encargar Webpack.

También podemos deshacernos de la opción *source map* y de la opción *module* ya que Webpack también nos ayuda con eso.

El archivo de configuración se puede quedar como aparece a continuación.

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "noImplicitAny": false
  }
}
```

Ahora vamos a irnos a configurar Webpack, y para eso vamos a crear un archivo `webpack.config.js`.

En este archivo vamos a declarar la ruta donde se encuentra nuestro código (**src**), y la ruta donde se encuentra el código final (**dist**).

Además tendremos que crear un módulo y exportarlo con la configuración de Webpack.

webpack.config.js

```
const path = require('path');
let entryPath = path.join(__dirname, 'src'),
      outputPath = path.join(__dirname, 'dist');

module.exports = {
```

```
}
```

Lo primero de todo es definir la propiedad **entry** donde se indica cual es el archivo principal de nuestra aplicación, que en este caso va a ser el `app.ts`.

webpack.config.js

```
const path = require('path');
let entryPath = path.join(__dirname, 'src'),
      outputPath = path.join(__dirname, 'dist');

module.exports = {
  entry: path.join(entryPath, 'ts/app.ts')
}
```

Lo siguiente que vamos a hacer es definir la ruta (**output.path**) y el archivo (**output.filename**) con todo el código de la aplicación (`bundle.js`) que tiene que generar Webpack y que será el vamos a importar en la página HTML.

webpack.config.js

```
const path = require('path');
let entryPath = path.join(__dirname, 'src'),
      outputPath = path.join(__dirname, 'dist');

module.exports = {
  entry: path.join(entryPath, 'ts/app.ts'),
  output: {
    filename: 'bundle.js',
    path: outputPath
  }
}
```

Como nuestro código se encuentra en Typescript, tenemos que pasarlo de alguna forma a JavaScript antes de generar el archivo `bundle.js`. Aquí es donde entran los **loaders** que se encargan de realizar transformaciones en los archivos, y en este caso vamos a utilizar el **ts-loader** que transformará el código Typescript a código JavaScript.

En este caso, los *loaders* van a estar dentro del array **module.rules** y cada uno de ellos será un objeto JavaScript con las siguientes opciones:

- test: es una expresión regular que indica a que archivos hay que aplicar el loader.
- exclude: es una expresión regular que indica a que archivos no hay que aplicar el loader, aunque coincidan con la expresión regular anterior.
- use: es un array con los loaders que hay que aplicar.

webpack.config.js

```
const path = require('path');
let entryPath = path.join(__dirname, 'src'),
      outputPath = path.join(__dirname, 'dist');

module.exports = {
  entry: path.join(entryPath, 'ts/app.ts'),
  output: {
    filename: 'bundle.js',
    path: outputPath
  },
  module: {
    rules: [
      {
        test: /\.ts$/,
        exclude: /node_modules/,
        use: ['ts-loader']
      }
    ]
  }
}
```

A la hora de importar en los archivos que hemos creado algunos módulos, tendremos que importarlos añadiendo la extensión del archivo.

```
import { Clase } from 'archivo.ts';
```

Para evitar tener que añadir la extensión podemos añadir una opción más al archivo de configuración de Webpack que es **resolve.extensions**, donde añadiremos un array con aquellas extensiones que estamos usando en nuestro código.

Con esa opción, cuando Webpack se encuentre una importación, va a buscar el archivo añadiendo esas extensiones hasta que una de ellas sea correcta.

webpack.config.js

```
const path = require('path');
let entryPath = path.join(__dirname, 'src'),
      outputPath = path.join(__dirname, 'dist');

module.exports = {
  entry: path.join(entryPath, 'ts/app.ts'),
  output: {
    filename: 'bundle.js',
    path: outputPath
  },
  module: {
    rules: [
```

```
      {
        test: /\.ts$/,
        exclude: /node_modules/,
        use: ['ts-loader']
      }
    ]
  },
  resolve: {
    extensions: ['.ts', '.js']
  }
}
```

Hemos instalado la dependencia de **webpack-dev-server** con la que vamos a levantar nuestra aplicación en un servidor local. Por último vamos a añadir la opción de **devServer.contentBase** donde vamos a indicar en que ruta se encuentran los archivos que nos tiene que servir.

webpack.config.js

```
const path = require('path');
let entryPath = path.join(__dirname, 'src'),
      outputPath = path.join(__dirname, 'dist');

module.exports = {
  entry: path.join(entryPath, 'ts/app.ts'),
  output: {
    filename: 'bundle.js',
    path: outputPath
  },
  module: {
    rules: [
      {
        test: /\.ts$/,
        exclude: /node_modules/,
        use: ['ts-loader']
      }
    ]
  },
  resolve: {
    extensions: ['.ts', '.js']
  },
  devServer: {
    contentBase: outputPath
  }
}
```

Ahora ya podemos crear nuestro archivo `app.ts` con el código de nuestra aplicación y el `index.html` con la página que nos va a servir *webpack-dev-server*.

dist/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>

</body>
<script src="bundle.js"></script>
</html>
```

src/app.ts

```
console.log('Hola mundo!!!');
```

Ahora para evitar tener que escribir el comando de webpack-dev-server con las opciones que le queramos añadir, vamos a crearnos un script de npm que lo haga por nosotros.

package.json

```
{
  "name": "ej-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "webpack-dev-server"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "ts-loader": "^4.2.0",
    "typescript": "^2.8.3",
    "webpack": "^4.6.0",
    "webpack-cli": "^2.1.2",
    "webpack-dev-server": "^3.1.4"
  }
}
```

Y ya podemos levantar nuestra aplicación con el comando:

```
$ npm start
```

24.3.1. Añadir jQuery

En caso de querer añadir una librería de JavaScript a la aplicación que tenemos configurada con Webpack, solo tenemos que instalar la librería:

```
$ npm install --save jquery
```

Y una vez que se ha instalado la tenemos que importar para evitar que nos de el error de que no encuentra la variable.

src/app.ts

```
import * as $ from 'jquery';

console.log('Hola mundo!!!');
$('body').html(`<h1>Hola mundo!!!</h1>`);
```