

The background of the slide is a dark field filled with numerous out-of-focus, colorful circles in shades of red, orange, yellow, green, blue, and pink, creating a bokeh effect. A solid red horizontal bar spans the entire width of the slide at the bottom.

TYPESCRIPT

TS para todo

TypeScript para todo

Alejandro Talaminos Barroso

Contenidos

1. Introducción
2. Cómo leer este libro
3. Fundamentos
 1. Introducción
 2. Primer programa en TypeScript
 3. Comando *tsc*
 4. Parámetros de comprobación estricta en TypeScript
 5. Parámetros de calidad de código en TypeScript
 6. Paquetes *@types*

7. Uso de TypeScript dentro de un proyecto Angular

4. Variables

1. Introducción
2. Tipos básicos
3. Tipos complejos
4. Tipos especiales
5. Detección de tipos
6. Ámbito de variables

5. Funciones

1. Tipado de funciones
2. *void* y *undefined* en las funciones
3. Tipar callbacks
4. Sobrecarga

6. Clases

1. Introducción

2. Modificadores de visibilidad *public* y *private*
3. Herencia
4. Modificador de visibilidad *protected*
5. Modificador *readonly*
6. Propiedades estáticas
7. Métodos *get* y *set*
8. Clases abstractas
9. Constructores privados y el patrón *singleton*

7. Interfaces

1. Introducción
2. Interfaces objeto
3. Interfaces clase
4. Interfaces función
5. Polimorfismo

8. Tipos avanzados

1. Tipo *intersection*
2. Tipo *guard*
3. Tipo *discriminated union*
4. Tipo *casting*
5. Tipos índice

9. Genéricos

1. Introducción
2. Creación de un tipo genérico
3. Restricciones en los genéricos
4. La partícula *keyof*
5. Tipos genéricos en clases
6. Tipos genéricos en interfaces
7. Tipo *Partial*
8. Tipo *Readonly*

10. Decoradores

1. Introducción
2. Decoradores de clases

3. Decoradores de atributos
4. Decoradores de accesores
5. Decoradores de métodos
6. Decoradores de parámetros en métodos
7. Orden de ejecución de los decoradores
8. Cambiando una clase con un decorador
9. Ejemplo de validación de los atributos de una clase con decoradores
10. Ejemplo de *autobind* con decoradores

11. Módulos

1. Modularización en TypeScript
2. Introducción a los módulos
3. Módulos en el frontend
4. Módulos en el backend
5. Funcionamiento de la exportación/importación

12. Otros

1. Manipulación del DOM con TypeScript
2. Librerías sin soporte para TypeScript
3. TypeScript con *Node* y *Express*
4. Otras herramientas

Introducción

TypeScript es uno de los lenguajes de programación que mayor auge ha experimentado en los últimos años. En el [informe anual](#) de GitHub, TypeScript se posiciona en la cuarta posición entre los lenguajes más importantes, tras JavaScript, Python y Java. La [comunidad de Stack Overflow](#) también sitúa a TypeScript en la segunda posición entre los lenguajes más apreciados, después de Rust. Por último, en una de las encuestas más importantes del ecosistema de JavaScript ([State of JavaScript](#)), donde participaron más de 23.000 programadores de 137 países del

mundo, TypeScript recibió el [premio a la tecnología más adoptada](#).

El éxito de TypeScript se ha materializado en importantes proyectos de software como Angular, Vue, Jest, Ionic, NativeScript, Deno, Yarn, RxJS, Visual Studio Code o GitHub Desktop. Grandes empresas del todo el mundo también han adoptado TypeScript como lenguaje en sus sistemas de desarrollo, entre las que se encuentran [Google](#), [Airbnb](#), [PayPal](#) o [Slack](#), entre otras muchas.

Asimismo, la comunidad de TypeScript crece constantemente y cada vez existe mayor cantidad de documentación en la red e incluso [repositorios de GitHub](#) muy completos con multitud de recursos relacionados con el lenguaje.

En este libro se presenta un recorrido profundo sobre las características más importantes del lenguaje TypeScript, desde los fundamentos más básicos, hasta conceptos avanzados relacionados con genéricos y decoradores. Dado que TypeScript extiende las funcionalidades de JavaScript, es recomendable que el lector posea conocimientos básicos de programación en este lenguaje, incluyendo la sintaxis básica y el manejo del entorno de ejecución [Node.js](#), junto al gestor de paquetes [npm](#).

Cómo leer este libro

El libro se encuentra organizado en doce capítulos con una complejidad creciente a medida que se avanza en la lectura. Cada capítulo se encuentra dividido en varias secciones de diversa extensión, en donde se expone abundante código descriptivo en prácticamente cada página.

La estructura y el enfoque del libro están orientados a un contenido predominantemente práctico, con el objetivo de que el lector pueda copiar, ejecutar en su computador y modificar los fragmentos de código expuestos. El editor de código recomendado para seguir este libro es [Visual Studio Code](#), de Microsoft,

incluyendo las siguientes extensiones de este editor y específicas para el lenguaje TypeScript: [Auto Import](#), [Auto Import - ES6, TS, JSX, TSX](#), [JavaScriptSnippets](#), [TypeScript Toolbox](#), [TypeScript Hero](#), [TypeScript's Getters and Setters Object Oriented Programming Style](#) y [Move TS - Move TypeScript files and update relative imports](#).

Las frases utilizadas durante todo el libro generalmente son concisas y directas. Se han evitado los párrafos largos de múltiples líneas para que el lector pueda centrarse en lo fundamental (el código), pero sin eludir las convenientes explicaciones que son necesarias. También existe material adicional en forma de enlaces a páginas web para profundizar en conceptos importantes que pueden ser de interés para el lector.

El código de TypeScript expuesto en este libro se encuentra claramente identificado con una tipografía distinta y con colores que resaltan la sintaxis de TypeScript. A continuación, se presenta un ejemplo:

```
const numero1 = 4;  
const numero2 = 5;  
const resultado = numero1 + numero2;  
  
console.log(`El resultado es ${resul
```



Los errores de código intencionados que son expuestos para acompañar a las explicaciones se encuentran debidamente indicados como comentarios. Por ejemplo, en la última línea del siguiente fragmento se presenta un error de código al realizar una asignación no válida de un valor numérico a una variable de tipo string.

```
// saludo es una variable de tipo st  
let saludo = 'hola';
```

```
// Error porque se asigna un valor n  
// saludo = 0;
```



En ocasiones, la primera línea de un fragmento de código indica el nombre del archivo que el lector debería de crear y, a continuación, el código que se presenta debería ser el contenido del archivo. Por ejemplo, en la primera línea del siguiente fragmento aparece el nombre del archivo *main.ts*, que debería crearse y contener la línea de código siguiente:

```
console.log('¡Hola mundo!');
```

```
// main.ts
```

```
console.log('¡Hola mundo!');
```

Los bloques de código están limitados a 80 caracteres por línea, siguiendo la regla [80/24](#), que es la recomendada en lenguajes como TypeScript.

Por último, es conveniente aclarar que el libro no está dirigido exclusivamente a los programadores de JavaScript que quieran introducirse al lenguaje TypeScript, sino también puede servir a otros programadores más experimentados como manual de referencia y consulta.

Para cualquier duda, sugerencia, notificación de errata o comentario pueden dirigirse al siguiente correo electrónico: librosatb@gmail.com

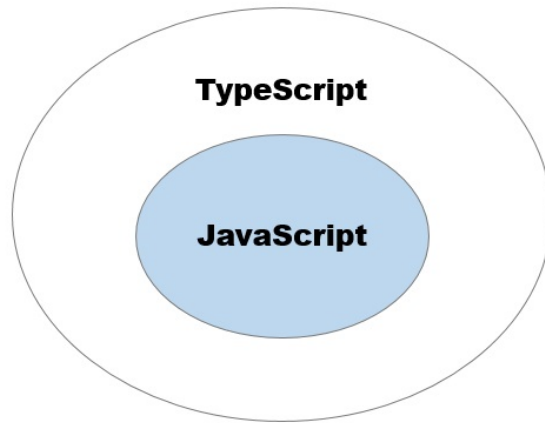
Muchas gracias por la compra de este libro.

Fundamentos

- [Introducción](#)
- [Primer programa en TypeScript](#)
- [Comando *tsc*](#)
- [Parámetros de comprobación estricta en TypeScript](#)
- [Parámetros de calidad de código en TypeScript](#)
- [Paquetes *@types*](#)
- [Uso de TypeScript dentro de un proyecto Angular](#)

Introducción

- **TypeScript** es un lenguaje de programación de código abierto y gratuito desarrollado por Microsoft.
- Los programas escritos en TypeScript no pueden ejecutarse directamente, sino que necesitan ser compilados al lenguaje JavaScript.
- Toda la sintaxis de JavaScript está soportada por TypeScript. Adicionalmente, TypeScript añade un conjunto de características no disponibles en JavaScript, como el tipado.
- En definitiva, TypeScript es un superconjunto (*superset*) de JavaScript.



- Algunas ventajas que proporciona TypeScript con respecto a JavaScript son:
 - Mejor auto-completado en editores de código que soportan TypeScript.
 - Clases y módulos más completos.
 - Detección cuando una variable no está declarada.
 - Detección cuando un objeto no posee una determinada propiedad.
 - Detección de cómo trabaja una función (parámetros de entrada, salida y tipos).

- Detección de errores en la sintaxis del lenguaje.
- Detección de malas prácticas escribiendo código.
- El tipado de TypeScript permite evitar comportamientos no esperados en la ejecución del código.

*/**

Código en JavaScript

**/*

```
function sumar(numero1, numero2) {  
    return numero1 + numero2;  
}
```

// Retorna 3

```
sumar(1, 2);
```

```
/*
```

Retorna '12' (comportamiento no es realiza una concatenación en lugar parámetros de la función (numero1)

```
*/
```

```
sumar('1', 2)
```



- Evitar este tipo de problemas en JavaScript puede ser tedioso.

```
/*
```

Código en JavaScript

```
*/
```

```
function sumar(numero1, numero2) {
```

```
/*  
    typeof es una función de JavaScr  
    tipo del valor o variable que se  
*/  
  
if (typeof(numero1) !== 'number' |  
    return console.log('Al menos uno  
}  
return numero1 + numero2;  
}
```

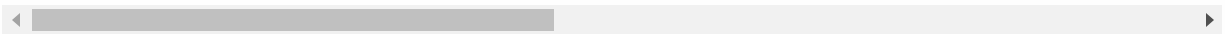
```
// Retorna 3  
sumar(1, 2);
```

```
/*  
  
Retorna undefined. La invocación a  
provoca que se cumpla el condicion  
ejecute el console.log y devuelva,
```

resultado de la invocación a conso

**/*

```
sumar('1', 2);
```



- El tipado en TypeScript permite que el código sea más claro y limpio.

*/**

Código en TypeScript

**/*

*/**

*Función que recibe dos parámetros
(tipo number) y retorna un valor d*

**/*

```
function sumar(numero1: number, nume  
    return numero1 - numero2;
```

```
}
```

```
// Retorna 3
```

```
sumar(1, 2);
```

```
/*
```

*Error en TypeScript porque la func
numérico*

```
*/
```

```
// sumar('1', '2')
```

```
/*
```

*Error porque TypeScript infiere qu
operación de concatenación dado qu
str2) y, por tanto, el valor resul
tipo string. En consecuencia, la f
valor de tipo number*


```
/*  
function concatenar (str1: string, s  
    return str1 + str2;  
}  
*/
```

Primer programa en TypeScript

- El primer paso es instalar TypeScript de forma global en el sistema (parámetro *-g*) con el gestor de paquetes *npm* de [Node.js](#). La instalación habilita el comando *tsc* (compilador de TypeScript) en la terminal de comandos del sistema operativo.

```
# Instala el comando tsc (compilador  
npm install -g typescript
```

- También es recomendable instalar TypeScript como dependencia de desarrollo en todos los proyectos software en los que se trabajen, dado que, si se actualiza TypeScript a nivel global, se mantienen las versiones locales de los proyectos.
 - Esto evita posibles problemas de incompatibilidades que puede acarrear una nueva versión de TypeScript en proyectos que ejecutan versiones antiguas.

Creación de un directorio para alo
`mkdir` nuevo-proyecto

Acceso al directorio creado anteri
`cd` nuevo-proyecto

Inicia un proyecto Node.js y crea

```
npm init
```

Instala TypeScript como dependenci

```
npm install typescript --save-dev
```



- Puede comprobarse la versión de TypeScript instalada en el sistema mediante la opción -v

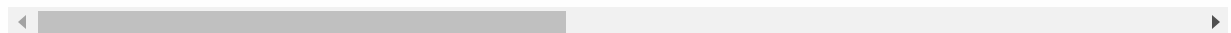
Versión global de typescript insta

```
tsc -v
```

Versión local de typescript instal

directorio principal del proyecto

```
npx tsc -v
```



- El siguiente paso es escribir un programa simple en TypeScript.

- Los archivos de código en TypeScript tienen la extensión *ts*.

```
// main.tsc
```

```
function saludar(nombre: string): string {  
    return "Hola, " + nombre;  
}
```

```
const saludo: string = saludar('Marc');  
console.log(saludo);
```



- A continuación, se compila el archivo de código mediante el comando *tsc* (TypeScript global) o el comando *npx tsc* (TypeScript local), seguido del nombre del archivo.

```
# compila con la versión de TypeScript  
tsc main.ts
```



```
# compila con la versión de TypeScript
```

```
npx tsc main.ts
```



- *tsc* genera un archivo de JavaScript con el mismo nombre que el archivo de TypeScript, pero con la extensión de los archivos de JavaScript (extensión *js*).

```
// main.js (código generado por tsc)
```

```
function saludar(nombre) {  
    return 'Hola, ' + nombre;  
}
```

```
var saludo = saludar('Marcos');
```

```
console.log(saludo);
```

```
console.log(saludo);
```



- Por defecto, TypeScript realiza una compilación a [ECMAScript 3 \(ES3\)](#) para garantizar la ejecución de código en todos los navegadores y entornos de ejecución de JavaScript.
- La página web oficial de TypeScript presenta un [editor](#) muy útil para comprobar en tiempo real cómo el código de TypeScript se convierte a JavaScript.

Comando *tsc*

- El comando *tsc* permite la compilación de múltiples archivos de TypeScript al mismo tiempo.

```
tsc archivo01.ts archivo02.ts archivo03
```

- También es posible monitorizar cambios en archivos de TypeScript mediante `tsc -w`. Esta instrucción permite compilar automáticamente todos los archivos de TypeScript cuando se produzcan cambios en alguno de ellos (por ejemplo, cuando se realice un guardado).
 - Para habilitar esta opción es necesario crear previamente el archivo de configuración de TypeScript, llamado `tsconfig.json`, dentro del directorio principal del proyecto (en el mismo nivel que el archivo `package.json`).

/* Ejemplo de archivo tsconfig.json

```
{
```

```
  "compilerOptions": {
```

```
    ..          ..    ..    ..
```

```
"target": "es5",  
"outDir": "dist",  
"rootDir": "src"  
}  
}
```



- En el archivo de configuración *tsconfig.json* se indican distintas opciones de compilación. Los principales [parámetros dentro de la propiedad *compilerOptions*](#) son:
 - *target*: versión de ECMAScript a compilar: *ES3* (valor por defecto), *ES5*, *ES6/ES2015*, *ES7/ES2016*, *ES2017*, *ES2018*, *ES2019*, *ES2020*, *ES2021*, *ES2022* y *ESNext* ([últimas novedades de JavaScript](#)).
 - *rootDir*: directorio donde se ubican los archivos de TypeScript.

- *outDir*: directorio donde se ubicarán los archivos de JavaScript tras la compilación. No se copiarán al directorio *outDir* los archivos contenidos en el directorio *rootDir* que no pueden ser compilados (por ejemplo, los archivos HTML o CSS).
- *module*: establece el sistema de módulos que se utilizará en la compilación. El valor por defecto es *commonjs*. Los módulos en TypeScript se describirán más adelante en el capítulo de Módulos.
- *lib*: es un parámetro de tipo array para poder establecer el conjunto de tipos específicos que pertenecen a **determinadas librerías de alto nivel de JavaScript** (DOM, Webworker, alguna versión específica de ECMAScript, ...). Este parámetro *lib* está muy relacionado con *target*. Asignar el valor *es6* a *target* incluirá

las siguientes opciones para el parámetro *lib*:
dom, *es6*, *dom.iterable* y *scripthost*.

- *allowJs*: parámetro booleano que permite incluir en la compilación los archivos de JavaScript.
- *checkJs*: parámetro booleano que permite verificar los posibles errores de sintaxis que puedan hallarse en los archivos JavaScript incluidos con el parámetro anterior (*allowJs*).
- *sourceMap*: parámetro booleano que ayuda a la depuración de proyectos de TypeScript. Genera un archivo con extensión *map* por cada archivo de JavaScript compilado. Estos archivos son entendidos por los navegadores web y ayudan a la depuración cuando se ejecuta el código (por ejemplo, desde la consola de Chrome).

- *removeComments*: parámetro booleano que permite eliminar o no los comentarios en los archivos JavaScript tras la compilación.
- *noEmit*: no realiza la compilación. Puede ser útil en grandes proyectos para aprovechar las funcionalidades del tipado de TypeScript sin necesidad de estar continuamente consumiendo recursos con la compilación.
- *noEmitError*: parámetro booleano que indica si los archivos compilados serán o no generados si existe cualquier error en los archivos TypeScript. El valor por defecto es *false*, es decir, se generarán los archivos siempre, incluso si existen errores.
- *strict*: parámetro booleano que permite o no realizar una comprobación estricta de los tipos. Si su valor es *true* (valor por defecto), entonces el resto de parámetros relacionados

con la comprobación estricta son establecidos a *true* (excepto si explícitamente se establecen a *false*).

- Fuera de la propiedad *compilerOptions* del archivo *tsconfig.json* también se pueden definir otras opciones:
 - *exclude*: array de archivos/directorios que pueden excluirse de la compilación, con la posibilidad de utilizar comodines o *wildcards*:
 - * (coincidencia de cero caracteres o más), ? (coincidencia de un carácter) y **/ (coincidencia recursiva en cualquier directorio).
 - El directorio *node_modules* es excluido por defecto si el parámetro *exclude* es omitido. En cambio, si es establecido,

entonces es fundamental añadir el directorio *node_modules*.

- *include*: array de archivos/directorios que son incluidos en la compilación. Si *include* es establecido, entonces solamente serán compilado los archivos definidos en este array, excepto los establecidos en el parámetro *exclude*.

/* tsconfig.json (ejemplo de archivo

```
{  
  "compilerOptions": {  
    /* ... */  
  },
```

/*

No se compilará el directorio no

**los archivos con nombres que ter
todos los archivos con nombres q
directorio interno**


```
*/  
"exclude": [  
  "node_modules",  
  "main1.ts",  
  "*.dev.ts",  
  "**/*.test.ts",  
]  
}
```



/* tsconfig.json (ejemplo de archivo

```
{  
  "compilerOptions": {  
    /* ... */  
  }  
}
```

```
},  
  
/*  
    Compila únicamente todo el conte  
    directorios internos)  
*/  
"include": ["src/**/*"],  
}
```



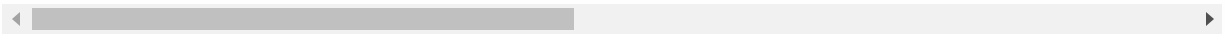
- Un archivo *tsconfig.json* predefinido puede crearse automáticamente mediante el comando *tsc*.

tsc --init

- Y posteriormente es posible ejecutar la instrucción *tsc -w* para monitorizar y compilar todos los archivos de TypeScript del directorio

actual, incluyendo los que se encuentran en subdirectorios.

```
# La consola se bloqueará hasta puls  
# en tiempo real los archivos de Typ  
tsc -w
```



- O simplemente compilar (todos los archivos del directorio actual y subdirectorios) y finalizar.

tsc

- En las siguientes secciones se presentan y describen algunos parámetros booleanos de compilación y calidad de código que pueden establecerse en el archivo *tsconfig.json* (dentro de la propiedad *compilerOptions*).

- Un posible ejemplo de archivo *tsconfig.json* se presenta a continuación, donde los archivos de TypeScript deben ubicarse en el directorio *src* y los de JavaScript (tras la compilación con *tsc*) se crearán en el directorio *dist*.


```
/* tsconfig.json */
```

```
{  
  "compilerOptions": {  
    "target": "ES2015",  
    "outDir": "dist",  
    "rootDir": "src",  
    "module": "commonjs",  
    "esModuleInterop": true,  
  }  
}
```


Parámetros de comprobación estricta en TypeScript

- *noImplicitAny*: permite o no declarar las variables con el tipo *any* implícitamente en los parámetros declarados de una función (donde no se puede garantizar el tipo que se recibe).

```
// Error si el parámetro noImplicitAny
function sumar(numero1, numero2) {
    return numero1 + numero2;
}
```



```
// Correcto si el parámetro noImplicit
function restar(numero1: number, numero2: number) {
    return numero1 - numero2;
}
```



- *strictNullChecks*: permite o no la comprobación de posibles valores nulos en las variables.

```
const boton1 = document.querySelector
```

```
/*
```

```
Error si strictNullChecks es true  
boton posee un valor o es null
```

```
*/
```

```
boton1.addEventListener('click', ()  
    console.log('Click')  
});
```

```
const boton2 = document.querySelector
```

```
/*
```

```
Correcto si strictNullChecks es tr  
onador de aserción no nulo (!) e
```

```

    operador de asignación no nulo (.) ~
    comprobación de valor nulo
*/

boton2.addEventListener('click', ()
    console.log('Click')
});

const boton3 = document.querySelector

/*
    Correcto si strictNullChecks es tr
    que verifica si la variable está d
*/

if (boton3) {
    boton3.addEventListener('click', (
        console.log('Click')
    })
}

```

J



- *strictBindCallApply*: permite que las propiedades *bind*, *call* y *apply* de las funciones estén tipadas y sean comprobadas mientras se escribe código.

```
function manejador(mensaje: string)  
    console.log(mensaje);  
}
```

```
const boton1 = document.querySelector
```

```
/*
```

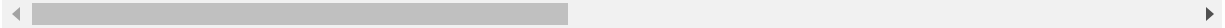
```
    Error si strictBindCallApply es tr  
    string
```

```
*/
```

```
boton1.addEventListener('click', man
```

```
const boton2 = document.querySelector
```

```
/*  
    Correcto si strictBindCallApply es  
    la función manejador  
*/  
boton2.addEventListener('click', man
```

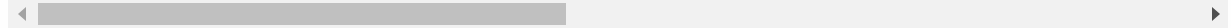


Parámetros de calidad de código en TypeScript

- *noUnusedLocals*: notifica de error si existen variables locales que no son utilizadas. No tiene efecto sobre variables de script, dado que TypeScript no puede saber si estas variables serán utilizadas por otros scripts.


```
function sumar(numero1: number, nume
```

```
// Error si noUnusedLocals es true  
const usuario = 'Alejandro';  
return numero1 + numero2;  
}
```




- *noUnusedParameters*: notifica de error si hay algún parámetro recibido por una función que no es utilizado.

```
/*  
Error si noUnusedParameters es true  
utilizado dentro de la función  
*/  
function sumar(numero1: number, nume  
    return numero1 + numero2;  
}
```



- *noImplicitReturns*: notifica de error si una función internamente posee un *return*, pero no devuelve valor para alguna de las rutas del flujo de código.

```
/*  
    Error si noImplicitReturns es true  
    ningún valor cuando a <= 0  
*/  
function sumar(a: number, b: number)  
    if (a > 0) {  
        return a + b;  
    }  
}
```



Paquetes @types

- Los paquetes [@types \(Definitely Typed\)](#) son una de las mayores fortalezas de TypeScript. Se trata de paquetes *npm* que añaden el tipado de TypeScript a paquetes de JavaScript.
- Por ejemplo, para utilizar [validator.js](#) con TypeScript puede instalarse el paquete de JavaScript (*validator*) y, seguidamente, el paquete que proporciona el tipado de TypeScript (*@types/validator*).

Instalación local del paquete vali

```
npm install validator
```

Instalación local como dependencia

el tipado para validator.js

```
npm install @types/validator
```



- A continuación, se crea el archivo de TypeScript y se importa el paquete instalado mediante la instrucción *import*. La importación/exportación de módulos se describirá más adelante en el capítulo de Módulos.

```
// main.tsc
```

```
// importación del paquete
```

```
import validator from 'validator';
```


```
/*
```

```
El autocompletado está ahora disponible para las propiedades disponibles  
(.) que se encuentra después del o de código y ayuda a no cometer errores  
aparecerán resaltadas en rojo)
```

```
*/
```

^/

```
validator.isEmail('prueba@mail.com')
```

◀  ▶

compilación del código

```
tsc main.tsc
```

ejecución del código

```
node main.js
```

- Antes que los *@types* se utilizaban los *tsd* y posteriormente los *Typings* (ambos están obsoletos a día de hoy).
- Para buscar el paquete *@types* asociado a un paquete de JavaScript puede accederse al [buscador de Microsoft](#). Aunque también se puede buscar directamente en Google o en el directorio *npm*. En general, el nombre del paquete será

siempre *@types/*, seguido del nombre del paquete en JavaScript.

- Algunos paquetes de JavaScript ya incluyen el tipado para TypeScript y no es necesario instalar su correspondiente paquete *@types*. Un ejemplo de este tipo de paquete es [mongodb](#).

Uso de TypeScript dentro de un proyecto Angular

- TypeScript es el lenguaje de programación utilizado para la construcción de aplicaciones frontend con el framework web Angular.
- Angular realiza automáticamente tareas como la compilación de código TypeScript, la generación

del archivo *package.json* y *tsconfig.json*, o el refresco automático de la página tras la modificación del código (tareas que en otras circunstancias tiene que realizarse de forma manual).

- Por tanto, Angular es una herramienta ideal para comenzar a aprender a TypeScript dado que proporciona todas las herramientas necesarias para comenzar a escribir código de la forma más simple y cómoda posible.

Instalación de Angular

```
npm install -g @angular/cli
```

Creación de un proyecto de Angular

Durante la creación del proyecto -

```
ng new ejemplo
```

```
# Acceso al directorio que contiene  
cd ejemplo
```

```
# Ejecución de la aplicación  
ng serve --open
```



- La aplicación de Angular estará disponible en la dirección <http://localhost:4200>
- El código de TypeScript puede comenzar a escribirse en el archivo
./src/app/app.component.ts

Variables

- Introducción
- Tipos básicos
- Tipos complejos
 - Tipo *object*
 - Tipo array
 - Tipo tupla
 - Tipo *enum*
 - Tipo *any*
- Tipos especiales
 - Tipo *union*
 - Tipo *literal*
 - Tipo *alias*

- Tipo *unknown*
- Tipo *void*
- Tipos *null* y *undefined*
- Tipo *never*
- Detección de tipos
- Ámbito de variables

Introducción

- Una de las ventajas importantes de TypeScript es la posibilidad de tipar las variables.
- Las variables en TypeScript se declaran con *const* (si la variable no modificará nunca su valor) o con *let* (si la variable modificará su valor posteriormente), seguido del tipo de variable y su asignación.

/*

*No es necesario tipar las variable
adelante la inferencia de tipos)*

*/

const numero1: **number** = 1;

let numero2: **number** = 2;



- A las variables declaradas con *const* se les debe asignar un valor obligatoriamente. No sucede lo mismo con las variables declaradas con *let*.

/*

*Error porque las variables declara
obligatoria de un valor*

*/

// *const* numero1: *number*;

/*

~ ~ ~ ~ ~

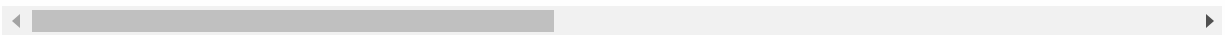
Correcto porque se declara la variable

**/*

```
let numero2: number;
```

```
numero2 = 2;
```

```
numero2 = 4;
```



- TypeScript puede inferir automáticamente el tipo de una variable a partir del valor que se le asigna, por lo que escribir el tipo es redundante. Por tanto, no es necesario establecer explícitamente el tipo de una variable cuando se puede inferir a partir de su valor.

```
const nombre = 'Mario Rodríguez';
```

- Visual Studio Code infiere el tipo automáticamente, aunque las variables declaradas con *const* las considera un tipo literal (un *string*

que no cambia nunca de valor), que es más estricto.

```
let nombre: string  
let nombre = 'Mario Rodríguez';
```

```
const nombre: "Mario Rodríguez"  
const nombre = 'Mario Rodríguez';
```

- En la declaración de las funciones, la inferencia de tipo automática no es posible, por lo que en este caso será necesario establecer el tipo para cada uno de los parámetros que recibe la función y el tipo del valor devuelto.


```
function sumar(numero1: number, numero2:  
    return numero1 + numero2;  
}
```



- No es posible cambiar el tipo de una variable una vez ha sido inferido, excepto si se tipa con *any* (o un tipo similar ambiguo).

```
// La variable saludo es de tipo str  
let saludo = 'hola';
```

```
// Error porque se asigna un valor n  
// saludo = 0;
```



Tipos básicos

- Tipo *boolean*: almacena exclusivamente valores *true* o *false*.

```
const estaEncendido = false;
```

- Tipo *number*: al igual que en JavaScript, en TypeScript solamente existe un tipo de variable numérica para almacenar valores enteros y flotantes, incluyendo también valores hexadecimales, binarios y octales.

```
const decimal = 6;  
const flotante = 3.5;  
const hexadecimal = 0xf00d;  
const binario = 0b1010;  
const octal = 0o744;
```

- Tipo *string*: TypeScript también permite utilizar comillas simples, dobles e invertidas (plantillas de cadena de texto) para definir variables de tipo *string*.
 - No existe consenso entre elegir comillas simples o dobles, si bien es cierto que [Google](#)

utiliza comillas simples en la documentación de Angular para incluir código HTML dentro de los *strings* (que pueden contener comillas dobles, como el valor de los atributos de las etiquetas HTML). En contraposición, Microsoft prefiere utilizar comillas dobles para los *strings*.

```
const nombre = 'Mario Rodríguez';  
const sentencia = `Hola, mi nombre e
```

// Las plantillas de cadena de texto

```
const sentencia: string = `Hola,  
    mi nombre es ${nombre}`;
```



Tipos complejos

Tipo *object*

- En TypeScript existen tres conceptos relacionados con los objetos que suelen dar lugar a confusión:
 - *Object*
 - *object*
 - `{}`
- *Object* permite realizar operaciones con objetos y, por supuesto, también está presente en JavaScript. Entre las propiedades de *Object* se encuentran: *Object.keys()*, *Object.values()*, *Object.freeze()*, etc.

/*

Código de JavaScript

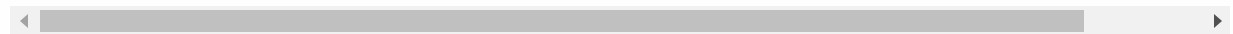
*/

```
const objeto = {  
  numero1: 1,  
  numero2: 2,  
  numero3: 3  
};
```

```
const propiedades = Object.keys(objeto)  
const valores = Object.values(objeto)
```

```
// ['numero1', 'numero2', 'numero3']  
console.log(propiedades);
```

```
// [1, 2, 3]  
console.log(valores);
```



- *object* y `{}` representan el mismo concepto en TypeScript. Una variable se puede tipar con estos

tipos, pero solamente tendrán accesible las propiedades de los objetos nativos de JavaScript (*toString*, *valueOf*, *hasOwnProperty*, etc..), con independencia de las propiedades que se establezcan en el objeto. En la práctica, esto es poco útil.

- Por tanto, *object* y `{}` permiten crear objetos vacíos, sin propiedades adicionales.

```
const objeto1: object = {  
  numero1: 1,  
  numero2: 2,  
};
```

```
const objeto2: {} = {  
  numero1: 1,  
  numero2: 2,  
};
```

```
/*  
    Error porque objeto1 y objeto2 est  
    respectivamente. Estos tipos no po  
*/  
// console.log(objeto1.numero1);  
// console.log(objeto2.numero1);
```



- En lugar de tipar con objetos genéricos, es preferible siempre ser más específico y tipar todas las propiedades (una a una).

```
const objeto: {  
    numero1: number;  
    numero2: number;  
} = {  
    numero1: 1,  
    numero2: 2,
```

```
};
```

```
console.log(objeto.numero1);
```

- Aunque la mejor práctica es utilizar la inferencia de tipos comentada con anterioridad. En este caso, los tipos se infieren automáticamente a partir de la asignación de los valores de las propiedades.

```
const objeto = {  
  numero1: 1,  
  numero2: 2,  
};
```

```
console.log(objeto.numero1);
```

- Sin embargo, cuando el objeto contiene valores donde no se puede inferir el tipo (por ejemplo, un

array vacío), será necesario establecer los tipos para cada una de las propiedades del objeto.

```
const objeto: {  
  numero1: number;  
  saludo: string;  
  numeros: number[];  
} = {  
  numero1: 1,  
  saludo: 'hola',  
  numeros: []  
};
```

```
objeto.numeros.push(2);
```

- En TypeScript no puede definirse un objeto vacío y, a continuación, agregar propiedades de forma dinámica (excepto si se declara como *any* o se

utiliza el tipo genérico *Partial*, que se describe en el capítulo de Genéricos).

```
const objeto1 = {  
  mostrarHola: () => {  
    console.log("Hola");  
  }  
};
```

```
// Error porque no se pueden agregar  
// objeto1.numero = 1;
```

```
const objeto2: any = {};
```

```
/*
```

```
Correcto porque objeto2 se ha decl  
propiedad agregada de forma dinámi
```

```
*/
```

```
objeto2.numero = 1;
```

- Sin embargo, se pueden declarar propiedades de un objeto como opcionales (utilizando el carácter ?). El valor de una propiedad opcional puede ser asignado posteriormente.

```
const objeto: {  
  numero1: number  
  // La propiedad numero2 se declara  
  numero2?: number;  
} = {  
  // No es necesario asignar un valor  
  numero1: 2  
};
```

```
// undefined  
console.log(objeto.numero2);
```

```
// Correcto porque la propiedad nume  
objeto.numero2 = 1;
```

```
// 1
```

```
console.log(objeto.numero2);
```



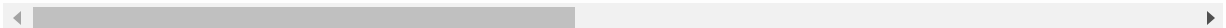
- Otra forma más recomendada de tipar objetos es utilizando clases o interfaces (preferiblemente ésta última), tal y como se describirá en capítulos posteriores.

Tipo array

- Los arrays en TypeScript se tipan escribiendo el tipo de los elementos que contiene y, a continuación, los corchetes.

```
// El array lista1 solamente puede a  
const lista1: number[] = [1, 2, 3];
```

```
// Otra forma de declarar un array q  
const lista2: Array<number> = [1, 2,
```



- Si el array se define sin tipo, entonces se pueden almacenar valores de distinto tipo considerando los valores asignados inicialmente.

```
/*
```

```
El tipado de cosas1 será inferido  
carácter | representa el tipo unio  
cosas1 solamente puede almacenar v
```

```
*/
```

```
const cosas1 = [true, 1, 'baloncesto
```

```
/*
```

```
Enfer porque cosas1 únicamente pue
```



```
Error porque cosas1 unicamente pue  
y string. No puede almacenar objet  
*/
```

```
// cosas1[0] = {}
```

```
// Correcto porque los valores de ti  
cosas1.push(true);
```

```
/*  
    cosas2 sí puede contener valores d  
    recomendable)  
*/
```

```
const cosas2: any[] = [true, 1];  
cosas2[0] = [];  
cosas2.push({});
```



Tipo tupla

- Una tupla es un array donde se definen los tipos de los valores permitidos por posición.
- En la asignación de una tupla es obligatorio que el número de valores y los tipos coincidan con los establecidos en la declaración.

*/**

*Declaración de una tupla de tres t
segundo es tipo number y el tercer*

**/*

```
let tupla: [string, number, boolean]
```

*/**

*Correcto porque los tipos de los v
orden: string, number y boolean*

```
*/
```

```
tupla = ['Hola', 10, true];
```

```
// Error porque el tercer valor es d
```

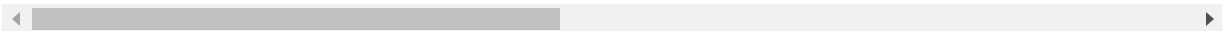
```
// tupla = ['Hola', 22, 3];
```

```
// Error porque se asignan dos valor
```

```
// tupla = ['Hola', 1];
```

```
// Error porque se asignan cuatro va
```

```
// tupla = ['Hola', 22, true, true];
```



- Pueden utilizarse los métodos de los arrays de JavaScript para insertar o eliminar valores (*push* y *pop*) en una tupla, pero solamente pueden agregarse valores de los tipos que son declarados

en la tupla. Esto sucede también con los arrays (excepto en arrays vacíos).

```
// Declaración de una tupla
```

```
const tupla: [string, number] = ['Ho
```

```
// Correcto
```

```
tupla.push('Adiós');
```

```
tupla.push(12);
```

```
// [ 'Hola', 10, 'Adiós', 12 ]
```

```
console.log(tupla);
```

```
/*
```

```
Error porque el tipo boolean no es  
valores de tipo string o number)
```

```
*/
```

```
// tupla.push(true);
```

// Declaración de un array que solo

const array1 = [1, 2];

// Correcto porque se inserta un va

array1.push(2);

// Error porque solamente se pueden

// array1.push('Hola');

// Declaración de un array vacío


const array2 = [];

*/**

*Correcto porque el array está vací
valor)*

**/*

```
array2.push(2);  
array2.push('Hola');  
array2.push(true);
```



- Por tanto, en la asignación de valores a una variable de tipo tupla se deben especificar tantos valores como los establecidos en la declaración, pero con posterioridad el número de elementos puede variar si se utilizan métodos de los arrays que modifican la longitud (como *pop* o *push*, entre otros).
- Si se invoca a un método que no existe para el tipo definido, entonces se produce un error.

```
const tupla: [string, number] = ['Ho
```

```
// Correcto
```

```
console.log(tupla[0].substr(1));
```

```
/*
```

```
Error porque el tipo number no pos  
strings)
```

```
*/
```

```
// console.log(tupla[1].substr(1));
```



- TypeScript arrojará un error cuando se acceda a la posición de un elemento que no existe en la tupla. No sucede lo mismo con los arrays, donde se devuelve *undefined*.

```
const tupla: [string, number] = ['Ho
```

```
// Error porque no existe valor para
```

```
// console.log(tupla[2]);
```

```
const array: number[] = [1, 2];
```

```
// undefined
```

```
console.log(array[2]);
```



- TypeScript no puede inferir automáticamente el tipo de una tupla y es necesario siempre declarar explícitamente los tipos de los elementos que contendrá.

Tipo *enum*

- Permite asignar nombres de variables más amigables a valores numéricos, de forma secuencial.

- Por defecto, el primer elemento de una enumeración comienza en 0 y los siguientes elementos toman los valores sucesivos.

// Declaración de un tipo enumeración

```
enum Color { Rojo, Verde, Azul }
```

// 0

```
console.log(Color.Rojo);
```

// 1

```
console.log(Color.Verde);
```

// 2

```
console.log(Color.Azul);
```

// Declaración de una variable del tipo

```
const colorVerde: Color = Color.Verde
```

```
// 1
```

```
console.log(colorVerde);
```



- Las enumeraciones pueden comenzar con otro valor.

```
enum Color { Rojo = 1, Verde, Azul }
```

```
// 1
```

```
console.log(Color.Rojo);
```

```
// 2
```

```
console.log(Color.Verde);
```

```
// 3
```

```
console.log(Color.Azul);
```



- Es posible cambiar manualmente el valor de cada uno de los elementos de la enumeración.

```
enum Color { Rojo = 1, Verde = 2, Az
```

```
// 1
```

```
console.log(Color.Rojo);
```

```
// 2
```

```
console.log(Color.Verde);
```

```
// 4
```

```
console.log(Color.Azul);
```



- También se pueden asignar valores de tipo *string* a los elementos de la enumeración. El elemento

inmediatamente posterior al asignado a un *string*
se le debe obligatoriamente asignar un valor.

```
/*
```

```
El elemento Azul obligatoriamente  
anterior es de tipo string
```

```
*/
```

```
enum Color { Rojo = 1, Verde = 'verd
```

```
// 1
```

```
console.log(Color.Rojo);
```

```
// verde
```

```
console.log(Color.Verde);
```

```
// 400
```

```
console.log(Color.Azul);
```

```
// 401
```

```
console.log(Color.Negro);
```



- Puede accederse al nombre del elemento de la enumeración utilizando el valor numérico asociado al elemento. Si el valor asociado al elemento de la enumeración es un string, entonces se devuelve *undefined*.

```
enum Color { Rojo = 1, Verde, Azul =
```

```
/*
```

```
No se trata del elemento ubicado e  
al valor 2 en la declaración de la  
Verde
```

```
*/
```

```
console.log(Color[2]);
```

```
// undefined  
console.log(Color['azul']);
```



Tipo *any*

- La variable que es declarada como *any* (supertipo universal) acepta cualquier tipo de valor. Es preferible evitar su uso siempre que sea posible.

```
let cajonDeSastre: any = 4;  
cajonDeSastre = '¡Hola mundo!';  
cajonDeSastre = false;
```

- Puede pensarse que el tipo *any* es similar al tipo *object*, pero no es cierto. Una variable de tipo *any*

es mucho más permisiva y TypeScript no realiza ninguna comprobación sobre sus propiedades.

```
const cualquiera: any = 4;
```

```
const objeto: object = {};
```

```
/*
```

```
Correcto sintácticamente porque el  
tipo any y TypeScript no realizará  
se producirá un error cuando se ej
```

```
*/
```

```
// cualquiera.siExiste();
```

```
/*
```

```
Correcto sintácticamente porque el  
tipo any, pero puede producirse un  
número, dado que toFixed es un mét
```

```
*/
```

```
cualquiera.toFixed();
```

```
// Error porque el método toFixed no
```

```
// objeto.toFixed();
```



- El tipo *any* puede ser útil para definir un array de elementos de distinto tipo.

```
const lista: any[] = [1, true, 'Hola
```

```
lista[1] = 'Adiós';
```



Tipos especiales

Tipo *union*

- El tipo *union* permite declarar variables que admiten varios tipos. Para ello se hace uso del carácter |

```
// La variable numero0String admite  
let numero0String: number | string =
```

```
// Correcto porque la variable admit  
numero0String = 'hola';
```

```
// Error porque la variable no admit  
// numero0String = true;
```



- También se puede utilizar el carácter | para definir un array que almacene diferentes tipos de valores.

```
const valores: (number | boolean)[ ]
```



- Aunque la inferencia de tipos también detecta automáticamente que se trata de una variable de tipo array con tipos de datos específicos.

```
// TypeScript inferirá el tipo (number | boolean)  
const valores = [1, 2, 3, true];
```

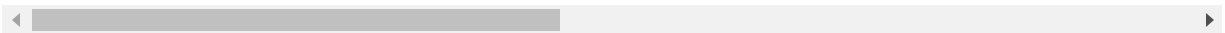
- TypeScript puede tener problemas para inferir determinadas operaciones en tipos ambiguos como *union*.

```
function concatenar(valor1: number | string, valor2: number | string): number | string {  
    /*  
       Error porque TypeScript desconoce la operación  
       concatenado (solamente los valores de tipo string se pueden concatenar)  
    */  
    // return valor1 + valor2;  
}
```

- Sin embargo, la función *typeof* puede ser utilizada para que TypeScript infiera los tipos ambiguos de *union* en determinados bloques de código.

```
function concatenar(valor1: number |  
  
if(typeof(valor1) == "number" && t  
  
    /*  
        El método toString permite con  
        los valores pueden ser concate  
    */  
  
    return valor1.toString() + valor  
}  
  
else if(typeof(valor1) == "string"  
  
    // Las dos variables son tipo st
```

```
    return valor1 + valor2;
  }
}
```



Tipo *literal*

- El tipo *literal* permite que una variable sólo admita valores específicos previamente establecidos en la declaración.

// La variable color únicamente pued

```
let color: 'rojo' | 'verde' | 'azul'
```

// Correcto

```
color = 'rojo';
```

```
color = 'verde';
```

```
/*
```

```
Error porque el amarillo no es una  
declaración de la variable
```

```
*/
```

```
// color = 'amarillo';
```



Tipo *alias*

- El tipo *alias* permite crear tipos personalizados mediante la palabra reservada *type* de TypeScript, pudiendo utilizar también los tipos *union* y literales de forma combinada.

```
/*
```

```
Creación de un tipo personalizado
```

```
tipo string number y el literal t
```

```
    tipo string, number y el literal true
*/
```

```
type Combinado = string | number | t
```

```
// Declaración de una variable de tipo
```

```
let cosa: Combinado;
```

```
// Correcto porque esta variable admite
```

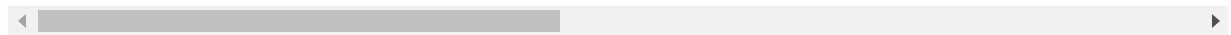
```
cosa = 'rojo';
```

```
cosa = 2;
```

```
cosa = true;
```

```
// Error porque la variable no admite
```

```
// cosa = false;
```



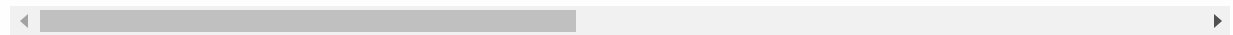
- El tipo *alias* permite crear tipos personalizados de objetos.

// Declaración de tipo personalizado

```
type Usuario = {  
  nombre: string;  
  edad: number;  
};
```

// Declaración de una variable tipada

```
const usuario: Usuario = {  
  
  nombre: 'Marcos',  
  edad: 23  
};
```



Tipo *unknown*

- El tipo *unknown* es similar a *any*, pero más restrictivo, lo que lo convierte en un tipo más

recomendable que *any*.

- Al igual que *any*, también permite almacenar valores de cualquier tipo.
- Sin embargo, la asignación de una variable de tipo *unknown* a una variable de otro tipo más restrictivo (*string*, *boolean*, etc.) genera un error.

```
let desconocido: unknown;
```

```
let cualquiera: any;
```

```
let str: string;
```

```
desconocido = 'Hola';
```

```
cualquiera = 4;
```

```
str = 'Adiós';
```

```
/*
```

Correcto porque a la variable str

*(cualquier valor) y esto está perm
porque se almacenará el valor 4 en
/

`str = cualquiera;`

*/**

*Se muestra undefined porque str es
ejecute el código y los números no
/

`console.log(str.length)`

*// Error porque a la variable str se
// str = desconocido;*

*/**

*Si se pretende realizar la asignac
necesario comprobar que la variabl
que el compilador de TvneScrint no*

```
    que es compatible de tipo any
*/
if (typeof(desconocido) === 'string')
    str = desconocido;
}
```

- Una variable de tipo *unknown* solo puede ser asignada a otra variable de tipo *unknown* o a *any*, excepto si previamente se ha establecido alguna comprobación con *typeof* o *instanceof*. La comprobación será automáticamente detectada por el analizador sintáctico de TypeScript.

Tipo *void*

- Es el opuesto a *any* y generalmente se utiliza para definir funciones que no retornan ningún valor.

```
function saludar(): void {
```

```
function saludar(): void {  
    console.log('Esto es un saludo');  
}
```

```
function advertir(): void {  
    console.log('Esto es un mensaje de  
  
    // Correcto porque la función no r  
    return;  
}
```

- Es posible también utilizar el tipo *void* en variables, pero es poco práctico puesto que únicamente pueden asignarse los valores *undefined* o *null*.

```
// Correcto porque void acepta solam  
const noUsable1: void = undefined;
```

```
// Error porque se asigna una variab  
// const noUsable2: void = 1;
```

Tipos *null* y *undefined*

- Funcionan igual que en JavaScript, aunque es poco práctico declarar variables con estos tipos porque solamente pueden tomar estos valores.

```
const noDefinido: undefined = undefi  
const nulo: null = null;
```

- En versiones antiguas de TypeScript, otros valores con tipos distintos podían tomar los valores *null* y *undefined*, pero actualmente ya no está permitido.

```
let str: string = 'Hola';
```


```
let numero: number = 5;
```

```
// Error en versiones antiguas de Ty
```

```
str = undefined;
```

```
// Error en versiones antiguas de Ty
```

```
numero = null;
```



- TypeScript permite ser más estricto para evitar que a una variable no se le pueda asignar valores nulos. Para ello puede modificarse la configuración de TypeScript (archivo *tsconfig.json*), concretamente asignando el valor *false* al parámetro *strictNullChecks* (por defecto está establecido a *false*, excepto si el parámetro *strict* se establece a *true*).

Tipo *never*

- El tipo *never* es utilizado por funciones que no retornan o finalizan, es decir, funciones que contienen bucles infinitos o arrojan excepciones.

```
function arrojarError(mensaje: string) {  
    throw { mensaje };  
}
```

```
function bucleInfinito(): never {  
    while (true) { }  
}
```

```
// Error porque esta función finaliz  
// function funcionQueRetorna(): nev
```

```
// Error porque esta función finaliz  
// function funcionQueRetorna(): nev
```

```
// function funcionQueRetorna(): nev
```



Detección de tipos

- Para detectar el tipo de una variable puede utilizarse la función *typeof*, teniendo en cuenta que los valores devueltos son los mismos que en JavaScript (*boolean*, *number*, *bigint*, *string*, *object*, *function* y *undefined*) y que algunos tipos de variables devuelven resultados en principio no esperados, como son el caso de *null* y los arrays.

```
const persona = {  
  nombre: 'José',  
  mayorDeEdad: true,  
  edad: 23,  
  numeroGrande: 100n,  
  domicilio: null
```

```
    domicilio: null,  
    localidad: undefined,  
    aficiones: ['Fútbol', 'Baloncesto'  
    edadDeNacimiento: new Date(),  
    hola: () => {  
        console.log('hola');  
    }  
};
```

```
// string
```

```
console.log(typeof(persona.nombre));
```

```
// boolean
```

```
console.log(typeof(persona.mayorDeEdad));
```

```
// number
```

```
console.log(typeof(persona.edad));
```


// bigint

```
console.log(typeof persona.numeroGra
```

// object porque null es considerado

```
console.log(typeof persona.domicilio
```

// undefined

```
console.log(typeof persona.localidad
```

// object porque los arrays son cons

```
console.log(typeof persona.aficiones
```

// object

```
console.log(typeof persona.edadDeNac
```

// function

```
console.log(typeof persona.bolo));
```

```
console.log(typeof(persona.nota));
```

```
// object
```

```
console.log(typeof(persona));
```



- Para garantizarse que una variable es de tipo array puede utilizarse la propiedad *isArray* del objeto *Array*.

```
const numeros = [1, 2, 3];
```

```
// object
```

```
console.log(typeof(numeros));
```

```
// true
```

```
console.log(Array.isArray(numeros));
```



- También se puede utilizar *instanceof* para comprobar las variables de tipo *Array* y otros tipos basados en objetos, como las fechas (*Date*).

```
const numeros = [1, 2, 3];
```

```
const fecha = new Date();
```

```
// true
```

```
console.log(numeros instanceof Array
```

```
// true
```

```
console.log(fecha instanceof Date);
```



Ámbito de variables

- *let* y *const* funcionan igual que en ECMAScript 6 o superior con respecto al ámbito de las variables.

Su uso es recomendable respecto a *var* o a variables globales.

Funciones

- Tipado de funciones
- *void* y *undefined* en las funciones
- Tipar callbacks
- Sobrecarga

Tipado de funciones

- Las funciones en TypeScript mantienen la misma sintaxis que las funciones de JavaScript, exceptuando el tipado.

```
// Función tipada
```

```
function tipada(): void { }
```

```
function sumar1(numero1: number, num  
    return numero1 + numero2;  
}
```

```
// Variable que almacena una función  
const sumar2 = function(numero1: num  
    return numero1 + numero2;  
};
```

```
// Variable que almacena una función  
const sumar3 = (numero1: number, num
```

```
/*  
    Variable tipada con Function. Indi  
    función, sin importar el número de  
*/
```

```
let sumar4: Function;
```

```
// Error porque sumar4 solamente pue  
// sumar4 = 5;
```

```
sumar4 = (numero1: number, numero2:  
    return numero1 + numero2 + numero3  
}
```

```
/*
```

```
    Creación de una variable tipada qu  
    parámetros de tipo number y devolv
```

```
*/
```

```
let sumar5: (numero1: number, numero
```

```
sumar5 = (numero1: number, numero2:  
    return numero1 + numero2;  
}
```



- Es posible definir parámetros opcionales mediante el carácter '?'. Por defecto, todos los parámetros son obligatorios.

// Ejemplo de función con dos paráme

const construirNombre1 = (nombre: **st**

return `\${nombre} \${apellidos}`;

};

// Error porque solamente se pasa un

// const resultado1 = construirNombr

// Error porque se pasan tres paráme

// const resultado2 = construirNombr

// Correcto

const resultado3 = construirNombre1(


```
// Antonio Sánchez
```

```
console.log(resultado3);
```

```
/*
```

```
    Los parámetros opcionales no puede  
    se suministran, su valor es undefi
```

```
*/
```

```
const construirNombre2 = (nombre: st
```

```
    /*
```

```
        Retorna la concatenación de nomb  
        valor definido y, si no lo tiene  
        de la variable nombre
```

```
    */
```

```
        return apellidos ? `${nombre} ${ap  
};
```

```
// Correcto
```

```
// Correcto
```

```
const resultado4 = construirNombre2(
```

```
// Pedro
```

```
console.log(resultado4);
```

```
// Error porque la función acepta so
```

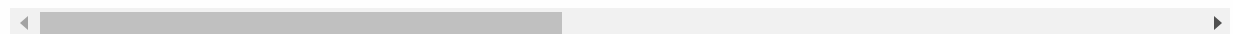
```
// const resultado5 = construirNombr
```

```
// Correcto
```

```
const resultado6 = construirNombre2(
```

```
// Pedro Ramírez
```

```
console.log(resultado6);
```



- También se pueden asignar valores por defecto a los parámetros de las funciones, convirtiéndolos

en opcionales.

```
const construirNombre1 = (nombre: st  
  return `${nombre} ${apellidos}`;  
};
```

// Correcto, el segundo parámetro es

```
const resultado1 = construirNombre1(
```

// Juan González

```
console.log(resultado1);
```

*/**

*Correcto. Recibe dos parámetros, t
defecto (González) dado que en la*

**/*

```
const resultado2 = construirNombre1(
```

```
// Juan González
```

```
console.log(resultado2);
```

```
// Error porque la función acepta so
```

```
// const resultado3 = construirNombr
```

```
// Correcto
```

```
const resultado4 = construirNombre1(
```

```
// Juan Ramírez
```

```
console.log(resultado4);
```

```
/*
```

```
Los parámetros con valores por def  
después de los parámetros obligato
```

```
*/
```

```
const construirNombre2 = (nombre) => {
```

```
const construirNombre2 = (nombre = ' ' ,  
    return `${nombre} ${apellidos}`;  
};
```

```
// Error porque la función espera do  
// const resultado5 = construirNombr
```

```
// Error porque la función espera do  
// const resultado6 = construirNombr
```

```
// Correcto
```

```
const resultado7 = construirNombre2(  

```

```
// María Prado
```

```
console.log(resultado7);
```

```
// Correcto
```

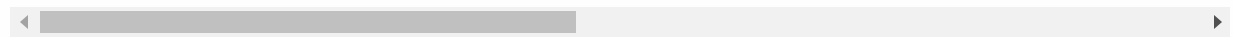
```
const resultado8 = construirNombre2(
```

```
// Ana Prado
```

```
console.log(resultado8);
```

```
// Error porque el primer parámetro
```

```
// const resultado9 = construirNombr
```



- Al igual que en JavaScript, las funciones en TypeScript pueden acceder a variables que están fuera del cuerpo de la función.

```
const numeroFuera = 100;
```

```
function sumar(numero1: number, nume  
    return numero1 + numero2 + numeroF  
}
```

```
// 103
```

```
console.log(sumar(1, 2))
```



- Por último, el número de parámetros que recibe una función no necesariamente tiene que ser fijo. Puede utilizarse el operador de propagación (...) para recibir un número indefinido de parámetros.

```
/*
```

```
La variable restoDeNombres es un a  
partir del segundo, que se le pasa
```

```
*/
```

```
const construirNombres = (nombre: st  
  return `${nombre} - ${restoDeNombr  
};
```

```
const nombres = construirNombres('Jo
```

```
// José - Lucía Lucas Paula
```

```
console.log(nombres);
```

void y *undefined* en las funciones

- Tanto en JavaScript como en TypeScript, las funciones que no retornan nada realmente devuelven *undefined* (incluso aunque estén declaradas para que retornen *void* en TypeScript).

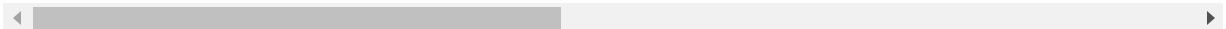
```
const noRetornarNada = (): void => {  
    const numero = 5;  
}
```

```
// undefined
```

```
console.log(noRetornarNada());
```


- Sin embargo, en TypeScript se producirá un error si se declara una función que retorne *undefined* si no incluye *return* o *return undefined* en el cuerpo de la función.

```
/*  
    Error porque la función mostrarSal  
*/  
/*  
const noRetornarNada = (): undefined  
    const numero = 5;  
}  
*/
```



- Será necesario incluir *return* dentro de la función para que sea sintácticamente válida para TypeScript.

```
const mostrarSaludo = (): undefined
```

```
console.log( 'Hola' );  
return;  
}
```

// Correcto

```
console.log(mostrarSaludo());
```



Tipar callbacks

- Los callbacks se tipan en TypeScript en la declaración de la función que la recibe.
 - Los callbacks generalmente son siempre el último parámetro que recibe una función.
 - Los callbacks generalmente no retornan nunca ningún valor (*void*).

*/**

```
/*
```

*El tercer parámetro de sumar es un
numérico y no retorna nada*

```
*/
```

```
const sumar = (  
  numero1: number,  
  numero2: number,  
  callback: (resultado: number) => v  
  
  const suma = numero1 + numero2;  
  callback(suma);  
);
```

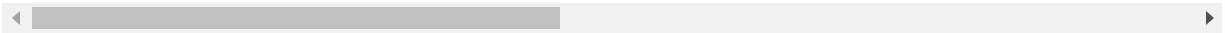
```
/*
```

*No es necesario tipar la variable
TypeScript la infiere del tipado d*

```
*/
```

```
sumar(1, 2, (resultado) => {
```

```
console.log(resultado);  
});
```



Sobrecarga

- Al contrario que en JavaScript, la sobrecarga en TypeScript es permitida.
- Para ello es necesario declarar la función tantas veces como se necesite, pudiendo cambiar el número y los tipos de los parámetros que recibe la función y el tipo del valor retornado. A continuación, es necesario implementar una función con todos los parámetros y el valor devuelto tipados a *any* o con un tipo combinado

que comprenda los tipos de todos los parámetros de todas las funciones a implementar.

- No se puede utilizar funciones de flecha para declarar funciones sobrecargadas. Es necesario emplear la sintaxis clásica de las funciones de JavaScript con *function*.

```
function sumarOConcatenar(numero: nu
```

```
function sumarOConcatenar(str: strin
```

```
function sumarOConcatenar(numero0Str
```

```
    return (typeof(numero0Str) === 'nu
```

```
        numero0Str + 2 :
```

```
        numero0Str + ' es un string';
```

```
}
```

```
// 6
```

```
console.log(sumarOConcatenar(4)).
```

```
console.log(sumarOConcatenar(true));
```

```
// Hola es un string
```

```
console.log(sumarOConcatenar('Hola'))
```

```
// Error porque la función sobrecarg
```

```
// sumarOConcatenar(true);
```



- Con la sobrecarga de funciones se puede evitar la ambigüedad de los tipos (por ejemplo, *any* o tipo *union*), si bien es cierto que será necesario diferenciarlos dentro de la implementación de la función.

Clases

- Introducción
- Modificadores de visibilidad *public* y *private*
- Herencia
- Modificador de visibilidad *protected*
- Modificador *readonly*
- Propiedades estáticas
- Métodos *get* y *set*
- Clases abstractas
- Constructores privados y el patrón *singleton*

Introducción

- El JavaScript clásico (antes de ECMAScript 6) utilizaba funciones y herencia basada en prototipos para construir código reutilizable.
- Sin embargo, los programadores de JavaScript nunca estuvieron cómodos con la sintaxis de los prototipos y demandaron un enfoque orientado a clases y objetos.
- A partir de ECMAScript 6, JavaScript incorporó las clases. TypeScript también las soporta y añade funcionalidades adicionales que existen en otros lenguajes de programación, como la posibilidad de definir propiedades públicas, privadas y protegidas.
- Las clases están compuestas por dos tipos de propiedades: los atributos (variables de la clase) y los métodos (funciones de la clase).

- El constructor de la clase es una función especial donde se inicializan los atributos de la clase durante la instanciación (creación del objeto a partir de la clase). Los constructores nunca retornan ningún valor.

```
class Persona {
```

```
// Atributos de la clase
```

```
nombre: string;
```

```
apellidos: string;
```

```
/*
```

```
Constructor de la clase, donde n
```

```
Los valores de estos parámetros
```

```
instanciación
```

```
*/
```

```
constructor(nombre?: string, apellidos?: string) {
```

```
CONSTRUCTOR (nombre?: string, apell
```

```
    // Se asignan los valores de los  
    this.nombre = nombre;  
    this.apellidos = apellidos;  
}
```

```
// Método de la clase  
getNombre() {
```

```
    // Se devuelve el valor del atri  
    return this.nombre;  
}
```

```
// Método de la clase  
getApellidos() {
```

```
    // Se devuelve el valor del atrib
```

```
        // se devuelve el valor del attr  
        return this.apellidos;  
    }  
}
```

```
const persona1: Persona = new Person  
const persona2: Persona = new Person
```

```
// undefined porque al atributo nomb  
console.log(persona1.getNombre());
```

```
// Alejandro  
console.log(persona2.getNombre());
```



- Para que una clase contenga varios constructores es necesario aplicar sobrecarga de funciones.

Modificadores de visibilidad

public y private

- Al igual que en otros lenguajes de programación, en TypeScript existen los modificadores de visibilidad *public* y *private* para definir el comportamiento de las propiedades de una clase.
- Si una propiedad se establece como *public*, entonces la propiedad puede ser accedida desde fuera de la clase.
- Si a un atributo no se le establece ningún tipo de modificador de visibilidad, entonces por defecto es *public*.

```
class Animal {
```

```
// Atributos declarados como públi
```

```
public nombre: string;
```

```
public edad: number;
```

```
constructor(nombre: string, edad:
```

```
    this.nombre = nombre;
```

```
    this.edad = edad;
```

```
}
```

```
// Método público
```

```
mover(distanciaEnMetros: number =
```

```
    console.log(`${this.nombre} se m
```

```
}
```

```
}
```


```
const serpiente = new Animal('Sammy'
```

```
/*
```

```
Correcto porque los dos atributos
```

```
públicos  
*/
```

```
console.log(serpiente.nombre);  
console.log(serpiente.edad);  
serpiente.mover(3);
```



- Por otra parte, las propiedades no pueden ser accedidas desde fuera de la clase si se declaran como *private*.

```
class Animal {
```

```
// Ahora el atributo nombre es pri  
private nombre: string;
```

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

```
}
```

```
private mover(distanciaEnMetros: n  
    console.log(`${this.nombre} se m  
}
```

```
public getNombre(): string {  
  
    return this.nombre;  
}  
}
```

```
const serpiente = new Animal('Sammy')
```

```
// Error porque el atributo es priva  
// console.log(serpiente.nombre);
```

```
// Error porque el método es privado  
// serpiente.mover(2);
```

```
// serpiente.mover(3);
```

```
// Correcto porque el método es públ  
console.log(serpiente.getNombre());
```



- En TypeScript es muy habitual simplificar la creación y asignación de valores a los atributos de una clase utilizando los modificadores de visibilidad (*private*, *protected* o *public*) junto a los parámetros que se establecen en el constructor.

```
class Persona {
```

```
    // private apellidos: string;
```

```
    /*
```

```
        Al utilizar un modificador de vi  
        automáticamente se crea un atrib
```

```
    */
```



```
constructor(nombre: string, privat  
  
mostrar() {  
  
    console.log(this.apellidos);  
  
    /*  
        Error porque nombre no es un a  
        parámetro del constructor y su  
        a esa función)  
    */  
    // console.log(this.nombre);  
}  
}
```

```
const persona = new Persona('Marcos')  
persona.mostrar();
```

Herencia

- Uno de los patrones fundamentales en la programación basada en clases es poder extender una clase existente para crear otras nuevas. Esto es conocido como herencia.
- Las clases que heredan (llamadas clases derivadas o subclasses) extienden una superclase o clase ascendente mediante la palabra clave *extends*.
- Una subclase hereda todos los atributos y métodos de la superclase que hereda.

// Animal es la superclase

```
class Animal {
```

```
    mover(distanciaEnMetros: number =  
        console.log(`El animal se movió`  
    )  
}
```

```
// Perro es la subclase que hereda d  
class Perro extends Animal {
```

```
    ladrar() {  
        console.log('¡Guau! ¡Guau!');  
    }
```


```
// Perro contiene también el método  
}
```

```
const perro = new Perro();
```

```
// Invocación del método ladrar de 1
```

```
// invocación del método ladrar de la
perro.ladrar();
```

```
// invocación del método mover de la
perro.mover(10);
```



- Las subclases pueden invocar desde su constructor a una función especial denominada *super()*, que se encarga de invocar al constructor de la superclase. Debe ubicarse obligatoriamente en la primera línea del constructor.

```
// Animal es la superclase
```

```
class Animal {
```

```
    constructor(private nombre: string
```

```
    mover(distanciaEnMetros: number =
        console.log(`El animal se movió
```

```
}

getNombre(): string {
    return this.nombre;
}
}

// Serpiente es la subclase que here

class Serpiente extends Animal {

    // El atributo nombre no es un atr
    constructor(nombre: string, privat

        // super invoca al constructor d
        super(nombre);
    }
}
```

-

```
const serpiente = new Serpiente('Sam
```

```
// Invocación a métodos de la superc
```

```
serpiente.mover();
```

```
console.log(serpiente.getNombre());
```



- Las subclases también pueden sobrescribir métodos heredados de la superclase.

```
class Animal {
```

```
    constructor(private nombre: string
```

```
    mover(distanciaEnMetros: number =
```

```
        console.log(`El animal se movió
```

```
    }
```

```
}
```

```
class Serpiente extends Animal {
```

```
    constructor(nombre: string) {
```

```
        // Invoca al constructor de la s
```

```
        super(nombre);
```

```
    }
```

```
    // Método sobrescrito (se encuentr
```

```
    mover(distanciaEnMetros: number =
```

```
        console.log('Reptando...');
```

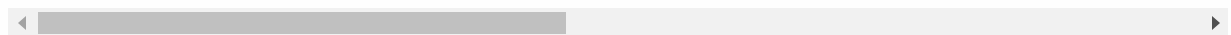
```
    /*
```

```
        super actúa aquí como una refe
```

```
        métodos de la superclase (en e  
        */  
        super.mover(distanciaEnMetros);  
    }  
}
```

```
const serpiente = new Serpiente('Sam
```

```
/*  
Invoca al método de la clase deriv  
mostrará:  
Reptando...  
El animal se movió 50 metros  
*/  
serpiente.mover(50);
```



- En TypeScript no está permitida la multiherencia (heredar de dos clases o más).

Modificador de visibilidad *protected*

- El modificador de visibilidad *protected* es utilizado en la herencia.
- El método o atributo con el modificador *protected* solamente puede ser accedido desde las clases derivadas.

```
class Animal {
```

```
  /*
```

```
    El atributo nombre (declarado co  
    clases derivadas, mientras que e
```

```
        accedido desde la clase Animal
    */

    constructor(protected nombre: string) {

        // el método mover es protegido
        protected mover(distanciaEnMetros:
            console.log(`${this.nombre} se m
        }

        public estaOculto() {
            return this.oculto;
        }
    }

    class Serpiente extends Animal {

        constructor(nombre: string) {
```

```
    super(nombre);  
}
```

```
public moverSerpiente(distanciaEnM  
    console.log('Reptando...');  
    this.mover(distanciaEnMetros);  
}
```

```
public getNombre(): string {
```

```
    /*
```

```
        Error porque el atributo oculto  
        desde la superclase Animal
```

```
    */
```

```
    // this.oculto = false;
```

```
    /*
```

*Correcto porque el atributo no
superclase y puede ser accedido
este caso)*

**/*

return this.nombre;

}

}

const serpiente = **new** Serpiente('Sam

*/**

*Error porque el atributo se encuen
ser accedido desde la clase que lo
(Serpiente)*

**/*

// console.log(serpiente.nombre);

```
// Error porque el método se encuent  
// serpiente.mover(2);
```

```
// Correcto
```

```
serpiente.moverSerpiente(3);
```

```
// Correcto
```

```
console.log(serpiente.getNombre());
```



Modificador *readonly*

- El modificador *readonly* permite definir un atributo de sólo lectura, donde las modificaciones únicamente pueden realizarse en el constructor.

```
class Animal {
```

```
// La asignación del valor del atributo nombre en el  
constructor(public readonly nombre  
  
setName(nombre: string) {  
  
// Error porque el atributo nombre es readonly  
// this.nombre = nombre;  
}  
  
getName(): string {  
    return this.nombre;  
}  
  
}  
  
// El atributo nombre toma el valor  
const animal = new Animal('Rocky');  
  
// Error porque el atributo nombre es readonly
```

```
// Error porque el atributo nombre es  
// animal.nombre = 'Nala';  
  
/*  
    Correcto porque, aunque el atributo  
    por tanto, puede accederse al mismo  
*/  
console.log(animal.nombre);
```

Propiedades estáticas

- Las propiedades estáticas son visibles a partir de la propia clase y pueden ser accedidas sin necesidad de crear ningún objeto con *new*.

```
class Animal {
```

```
    // Atributo privado estático
```

```
// Atributo privado estático  
private static nombre = 'Rocky';  
  
// Método público estático  
static mostrarNombre() {  
    console.log(Animal.nombre);  
}  
  
// Método público estático  
public static cambiarNombre(nombre)  
    Animal.nombre = nombre;  
}  
  
}  
  
// Error porque el atributo estático  
// console.log(Animal.nombre);  
  
// Rocky
```



```
// Rocky
```

```
Animal.mostrarNombre();
```

```
// Cambia el valor del atributo está
```

```
Animal.cambiarNombre('Nala');
```

```
// Nala
```

```
Animal.mostrarNombre();
```



- Las propiedades estáticas no pueden ser accedidas mediante *this* dentro de un método no estático.
- Sin embargo, y al contrario que sucede con otros lenguajes de programación, *this* sí puede utilizarse dentro de un método estático, donde apuntaría a la propia clase.

Métodos *get* y *set*

- TypeScript, al igual que JavaScript, aporta las partículas especiales *get* y *set*, seguido de un espacio y el nombre del atributo, para poder obtener y modificar los atributos de una clase desde fuera de la misma.
- Estos métodos *get* y *set* no necesitan ser invocados desde fuera de la clase (utilizando paréntesis), sino que pueden ser accedidos directamente como si fueran atributos de la clase.

```
class Coche {
```

```
    constructor(
```

```
        public posicion: number = 0,
```

```
/*
```

```
    Se añade un guión bajo para qu  
    y set asociados a este atribut
```

```
*/
```

```
private _velocidad: number = 42,  
private readonly MAXIMA_VELOCIDA
```

```
mover(): void {  
    this.posicion += this._velocidad  
}
```

```
// Los métodos de tipo get siempre
```

```
get velocidad(): number {  
    console.log('Obtiene la velocida  
    return this._velocidad;  
}
```

```
/*  
    Los métodos de tipo set siempre  
    retornar nada (no se puede utili  
*/  
  
set velocidad(value: number) {  
    console.log('Cambia la velocidad  
    this._velocidad = Math.min(value  
}  
}
```



```
const coche = new Coche();
```

```
/*  
    Se invoca al método get de velocid  
    La consola mostrará:  
        Cambia la velocidad  
*/
```

```
coche.velocidad = 120;
```

```
/*
```

```
    Se invoca al método set de velocidad
```

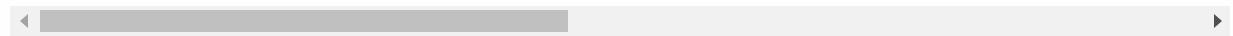
```
    La consola mostrará:
```

```
        Obtiene la velocidad
```

```
        100
```

```
*/
```

```
console.log(car.velocidad);
```



- Los métodos *get* y *set* son también llamados métodos accesoros.
- Una de las ventajas de estos métodos es permitir añadir una lógica de programación previa a la obtención/modificación de un atributo.

Clases abstractas

- Las clases abstractas son aquellas que generalmente incluyen al menos un método abstracto.
- Un método abstracto es aquel que no posee implementación (no tiene cuerpo). La implementación se escribe en una clase no abstracta que herede de una clase abstracta.
- Un método abstracto debe estar convenientemente tipado en la clase abstracta respecto a los parámetros que recibe y el tipo de valor que retorna.
- La palabra clave *abstract* se utiliza para declarar clases abstractas, así como también métodos abstractos dentro de una clase abstracta.

- A diferencia de las interfaces (ver capítulo de Interfaces), una clase abstracta puede contener métodos abstractos y no abstractos.
- No pueden crearse objetos a partir de una clase abstracta.

// La clase Animal es abstract

```
abstract class Animal {
```

// Método abstracto (sin implement

```
abstract hacerSonido(): void;
```

// Método no abstracto (con implem

```
mover(): void {
```

```
    console.log('Moviendo...');
```

```
}
```

```
}
```

```
/*
```

```
Esta clase está obligada a impleme  
hereda de la clase abstracta Anima
```

```
*/
```

```
class Vaca extends Animal {  
    hacerSonido() {  
        console.log( 'Muuuu' );  
    }  
}
```

```
// Error porque no se puede crear un  
// const animal: Animal = new Animal
```

```
// Correcto porque se crea un objeto
```

```
const vaca: Vaca = new Vaca();  
vaca.hacerSonido();
```



Constructores privados y el patrón *singleton*

- En ocasiones puede ser deseable que solamente pueda crearse una única instancia de una clase. Esto se consigue con el patrón *singleton* y los constructores privados.
- Si el constructor de una clase es declarado como privado, entonces no es posible crear ninguna instancia.

```
class Animal {
```

```
    // Constructor declarado como priv  
    private constructor(private nombre  
}
```

```
/*
```

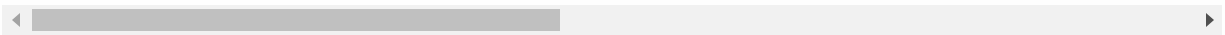
,

*Error porque no es posible crear i
es declarado como privado*

**/*

// const animal = new Animal('Sammy'

// const animal = new Animal();



- Con el uso de un método estático público que implemente la creación de instancias puede controlarse para que solamente pueda crearse una única instancia de una clase (patrón *singleton*). Para ello es necesario que exista un atributo privado estático dentro de la clase, que permita acceder a la única instancia de la clase que puede crearse.

class Animal {

// Instancia única que puede crear

```
private static instancia: Animal;
```

```
// Constructor declarado como priv
```

```
private constructor(private nombre
```

```
/*
```

```
Método público para obtener una  
clase
```

```
*/
```

```
public static obtenerInstancia() {
```

```
// Si la instancia no existe, se
```

```
if(!Animal.instancia) {
```

```
    Animal.instancia = new Animal(  
}
```

```
// Si la instancia ya existe, se
```

```
        return Animal.instancia;
    }

}

// Se obtiene una instancia de la cl
const animal1 = Animal.obtenerInstan

// { nombre: 'Sammy' }
console.log(animal1);

/*
    Se obtiene otra instancia de la cl
    que la anterior creada)
*/
const animal2 = Animal.obtenerInstan

// { nombre: 'Sammy' }
```

```
console.log(animal2);
```

```
// true porque animal1 y animal2 hac
```

```
console.log(animal1 === animal2);
```



Interfaces

- [Introducción](#)
- [Interfaces objeto](#)
- [Interfaces clase](#)
- [Interfaces función](#)
- [Polimorfismo](#)

Introducción

- Las interfaces son un mecanismo de la programación orientada a objetos que trata de suplir la carencia de la herencia múltiple. La

mayoría de los lenguajes no ofrecen la posibilidad de declarar una clase que herede de varias clases a la vez y, sin embargo, a veces es deseable. Por eso surgen las interfaces.

- La diferencia de las interfaces con respecto a las clases es que en las interfaces se declaran solamente métodos abstractos (métodos que no poseen implementación) y atributos sin valor (solamente se declara su tipo).
- En las interfaces no se utiliza la partícula *abstract* para definir métodos abstractos, al contrario que sucede en las clases abstractas.
- En definitiva, en las interfaces únicamente se declaran los atributos sin valor y los métodos tipados sin implementación.

Interfaces objeto

- Para crear una interfaz se utiliza la partícula *interface*.
- Los objetos que usen una interfaz deben poseer exactamente los mismos atributos y métodos que los definidos en la interfaz. Ni uno más, ni uno menos.

/*

Creación de una interfaz, con dos implementación

*/

```
interface PersonaInterfaz {  
    nombre: string;  
    apellidos: string;  
    mostrarNombre(): void;  
}
```



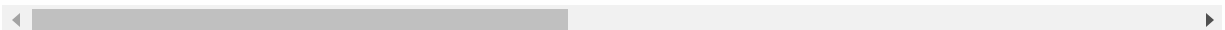
```
// Creación de un objeto tipado a pa  
const persona1: PersonaInterfaz = {  
  nombre: 'Alejandro',  
  apellidos: 'Pérez',  
  mostrarNombre(): void {  
    console.log(this.nombre);  
  }  
};
```

```
// Alejandro  
persona1.mostrarNombre();
```

```
// Error porque el atributo correoEl  
/*  
const persona2: PersonaInterfaz = {  
  nombre: 'Jorge',
```

```
apellidos: 'Pérez',  
correoElectronico: 'correo@mail.co  
mostrarNombre(): void {  
    console.log(this.nombre);  
}  
};  
*/
```

```
// Error porque faltan propiedades q  
/*  
const Persona3: PersonaInterfaz = {  
    nombre: 'Jorge'  
};  
*/
```



- También pueden establecerse propiedades opcionales en una interfaz mediante el carácter ?

```
/*  
    La interfaz CuadradoInterfaz defin  
    opcionales)  
*/  
interface CuadradoInterfaz {  
    color?: string;  
    ancho?: number;  
    mostrarAncho?(): void;  
}  
  
// El objeto cuadrado únicamente est  
const cuadrado: CuadradoInterfaz = {  
    color: 'rojo'  
};
```



- En las interfaces se pueden crear también atributos que son únicamente de lectura mediante

readonly. Sus valores no pueden ser modificados después de la asignación.

// La interfaz define dos atributos

```
interface Punto {  
    readonly x: number;  
    readonly y: number;  
}
```


```
const punto1: Punto = {  
    x: 10,  
    y: 20  
};
```

// Error porque el atributo y es de
// p1.y = 3;



- Las propiedades definidas en una interfaz no aceptan los modificadores *public*, *private* o *protected*.

```
/*  
    Error porque no se pueden utilizar  
    interfaz  
*/  
  
/*  
interface Punto {  
    private readonly x: number;  
    readonly y: number;  
}  
*/
```



- La interfaz puede definirse para que el objeto que lo implemente incorpore más propiedades de las que se establecen en la interfaz. Para ello se

utiliza la sintaxis de los corchetes en el nombre de la propiedad. Esto es conocido como Tipos índice y es descrito más adelante en el capítulo de Tipos avanzados.

```
/*
```

```
La interfaz define dos propiedades  
declarada indica que un objeto que  
propiedades (atributos o métodos),  
sus valores puede ser de cualquier
```

```
*/
```

```
interface CuadradoInterfaz {  
    color?: string;  
  
    ancho?: number;  
    [propiedadNombre: string]: any;  
}
```

```
/*
```

/

*El objeto establece un valor para
método no definido en la interfaz*

**/*

```
const cuadrado: CuadradoInterfaz = {  
  ancho: 100,  
  mostrarAncho() {  
    console.log(this.ancho);  
  }  
};
```



- Las interfaces objeto tienen **muchas similitudes con el tipo *alias*** (declarado con *type*). Aunque las **diferencias son mínimas**, las interfaces generalmente son utilizadas para describir la estructura de un objeto o de una clase, mientras que el tipo *alias* suele utilizarse para definir otros

tipos distintos a los objetos. Sin embargo, el tipo *alias* también puede utilizarse para tipar objetos.

```
type PersonaTipo = {  
    nombre: string;  
    apellidos: string;  
    mostrarNombre(): void;  
}
```

// Creación de un objeto a partir de

```
const persona: PersonaTipo = {  
    nombre: 'Alejandro',  
    apellidos: 'Pérez',  
    mostrarNombre(): void {  
        console.log(this.nombre);  
    }  
}  
  
persona.mostrarNombre();
```



```
persona.implements CoordnadasInterface {
```

Interfaces clase

- Para indicar que una clase implemente una interfaz se utiliza la partícula *implements*.
- La interfaz obliga a que la clase declare todos los atributos e implemente todos los métodos definidos en la interfaz.
- Al contrario que con los objetos, las clases que implementan una interfaz pueden añadir propiedades adicionales a las definidas en la interfaz.

```
// Interfaz que define dos atributos
```

```
interface CoordenadasInterface {
```

```
    x : number ;
```

```
x: number;  
y: number;  
pintar?(): void;  
}
```

// La clase Coordenadas implementa I
class Coordenadas **implements** Coorden

```
constructor(public x, public y) {
```

```
pintar() {  
    console.log(this.x, this.y);  
}
```

// Método no definido en la interf

```
getX() {  
    return this.x;  
}
```

```
    },  
}
```

```
const coordenadas: Coordenadas = new
```

```
// 2 3
```

```
coordenadas.pintar();
```

```
// 2
```

```
console.log(coordenadas.getX());
```



- Una clase puede implementar múltiples interfaces. En estos casos, la clase debe incluir todos los atributos e implementar todos los métodos incluidos en todas las interfaces implementadas.

```
interface PersonaInterfaz1 {
```

```
    nombre: string;  
}
```

```
interface PersonaInterfaz2 {  
    respirar(): void;  
}
```

```
/*  
    La clase Persona implementa dos in  
    PersonaInterfaz2  
*/
```


```
class Persona implements PersonaInte
```

```
/*  
    El atributo nombre debe establec  
    PersonaInterfaz1  
*/
```

```
constructor(public nombre: string)
```

```
/*  
    El método respirar debe establec  
  
    PersonaInterfaz2  
*/  
respirar(): void {  
    console.log('Respirando...');  
}  
}
```

```
const persona = new Persona('Alejand  
persona.respirar();
```



- Adicionalmente, una interfaz puede heredar de una o más interfaces.

```
interface PersonaInterfaz1 {  
    nombre: string;
```

```
    nombre = "Juan";  
}
```

```
interface PersonaInterfaz2 {  
  
    respirar(): void;  
}
```

```
/*  
    La interfaz PersonaInterfaz3 hereda  
    PersonaInterfaz2  
*/
```

```
interface PersonaInterfaz3 extends P
```

```
/*  
    La clase Persona implementa la interfaz  
    incluir todos los atributos y métodos de  
    PersonaInterfaz2  
*/
```

```
class Persona implements PersonaInte
```

```
    constructor(public nombre: string)
```

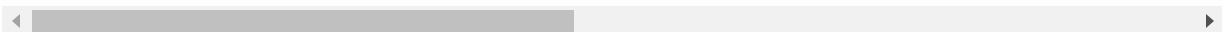
```
    respirar(): void {
```

```
        console.log('Respirando...');
```

```
    }
```

```
}
```

```
const persona = new Persona('Alejand  
persona.respirar();
```



Interfaces función

- Aunque es menos habitual, también pueden declararse interfaces función. En este caso, se debe declarar en una interfaz los nombres y tipos de los parámetros que recibirá la función y el tipo del valor retornado.

```
interface FuncionInterfaz {  
    (texto: string, cadena: string): b  
}
```

```
/*
```

```
    Función tipada a partir de una int  
    establecidos en la interfaz, inclu
```

```
*/
```

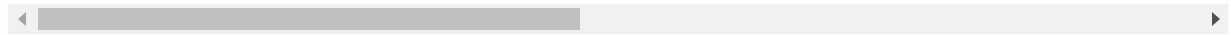
```
const buscar: FuncionInterfaz = (tex  
    return texto.search(cadena) > 1;  
}
```



- Sin embargo, para estos casos es más conveniente utilizar un alias con *type*.

```
type FuncionInterfaz = (texto: string
```

```
const buscar: FuncionInterfaz = (tex  
    return texto.search(cadena) > 1;  
}
```



Polimorfismo

- El polimorfismo permite que un objeto pueda invocar a métodos con igual nombre, parámetros y tipos, pero con distinta implementación.
- Cada implementación estará en una clase distinta, donde cada clase implementará la misma interfaz.

```
// Interfaz que implementarán todas  
interface ConectorInterfaz {  
    conectar(): boolean;  
}
```

```
class WifiConector implements Conect
```

```
/*
```

```
La clase WifiConector está oblig
```

```
implementa la interfaz ConectorI
```

```
*/
```

```
public conectar(): boolean {  
    console.log('Conectando vía wifi'  
    console.log('Estableciendo contr  
    console.log('Solicitando direcci  
    console.log('Conectado');  
    return true;  
}
```

```
}  
}
```

```
class BluetoothConector implements C
```

```
/*
```

```
La clase BluetoothConector está  
porque implementa la interfaz Co
```

```
*/
```

```
public conectar(): boolean {  
    console.log('Conectando vía Blue  
    console.log('Pareando con un PIN  
    console.log('Conectado');  
    return true;  
}  
}
```

```
class Sistema {
```

```
    /*
```

```
        El parámetro conector es de tipo  
        ejecución es un objeto de la cla
```

```
    */
```

```
constructor(private conector: Cone
```

```
    conectar() {
```

```
        this.conector.conectar()
```

```
    }
```

```
}
```

```
/*
```

```
    El atributo conector de la clase S  
    BluetoothConector
```

```
*/
```

```
const sistema1 = new Sistema(new Blu  
  
/*  
    El atributo conector de la clase S  
    WifiConector  
*/  
const sistema2 = new Sistema(new Wif  
  
// Se invoca al método conectar de l  
  
sistema1.conectar();  
  
// Se invoca al método conectar de l  
sistema2.conectar();
```



- Por tanto, las interfaces obligan a que las variables posean una estructura específica, pero no determinan directamente su tipo.

```
interface PersonaMostrarInterfaz {  
    mostrarNombre(): void;  
}
```

// La clase Persona no implementa ni

```
class Persona {  
  
    constructor(private nombre: string  
  
    mostrarNombre() {  
        console.log(this.nombre);  
    }  
}
```

*/**

*Se declara el objeto persona sin r
a que deba implementarse el método*

PersonaMostrarInterfaz

**/*

let persona: PersonaMostrarInterfaz;

*/**

*Correcto porque, aunque la clase P
PersonaMostrarInterfaz, sí incluye
el método mostrarNombre, que se en
PersonaMostrarInterfaz*

**/*

persona = **new** Persona('Marcos', 'Rod



Tipos avanzados

- Tipo *intersection*
- Tipo *guard*
- Tipo *discriminated union*
- Tipo *casting*
- Tipos índice

Tipo *intersection*

- El tipo *intersection* está estrechamente relacionado con el tipo *union*, pero se utilizan de forma distinta.

- Un tipo *intersection* combina varios tipos en uno solo y obliga a contenerlos a todos, mientras que el tipo *union* solamente obliga a contener uno de ellos.
- El tipo *intersection* utiliza el carácter &, mientras que el tipo *union* utiliza el carácter |

```
type PermisosAdministrador = {  
  servidor: boolean;  
  red: boolean;  
}
```

```
type PermisosProgramador = {  
  repositorio: boolean;  
  web: boolean;  
}
```

// Tipo intersection de los dos tipo

```
type PermisosGestorIntersection = Pe
```

```
/*
```

```
    Variable de tipo PermisosGestorInt
```

```
    todas las propiedades de los tipos
```

```
*/
```

```
const permisosGestor1: PermisosGesto
```

```
    servidor: true,
```

```
    red: true,
```

```
    repositorio: true,
```

```
    web: true
```

```
}
```

```
/*
```

```
    Error porque el tipo PermisosGesto
```

```
    propiedades definidas en los tipos
```

```
/*
```

```
/*  
const permisosGestor2: PermisosGesto  
    servidor: true,  
    red: true  
  
}  
*/
```



- A continuación, se presenta el mismo ejemplo pero utilizando el tipo *union*.

```
// Tipo union  
type PermisosGestorUnion = PermisosA  
  
/*  
    Correcto porque contiene todas las  
    (PermisosAdministrador)  
*/  
const permisosGestor1: PermisosGesto
```

```
const permisosGestor1: PermisosGesto
  servidor: true,
  red: true
}
```

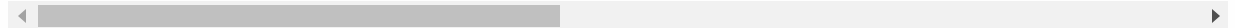
```
/*
  Correcto porque contiene todas las
  (PermisosProgramador)
*/
```

```
const permisosGestor2: PermisosGesto
  repositorio: true,
  web: true
}
```

```
/*
  Correcto porque contiene todas las
  (PermisosAdministrador y PermisosP
*/
```

```
const permisosGestor3: PermisosGesto
```

```
  servidor: true,  
  red: true,  
  repositorio: true,  
  web: true,  
}
```



- El tipo *union* e *intersection* se pueden utilizar conjuntamente.

```
type Combinado1 = string | number;  
type Combinado2 = number | boolean;  
type Universal = Combinado1 & Combin
```

```
/*
```

```
  La variable numero solamente puede  
  tipo Combinado1 y Combinado2)
```

```
*/
```

```
const numero: Universal = 4;
```



- El comportamiento del tipo *intersection* es similar a una interfaz que hereda de otras interfaces.

Tipo *guard*

- Los tipos *union* son útiles para modelar variables con tipos superpuestos.
- Sin embargo, en algunas situaciones es deseable conocer exactamente el tipo utilizado en una variable declarada como tipo *union*.
- El tipo *guard* permite limitar el tipo de una variable dentro de un bloque condicional.

```
type alfanumerico = string | number;
```

```
function sumarOConcatenar(var1: alfa
```

```
/*
```

```
    Tipo guard donde var1 y var2 son  
    bloque del condicional
```

```
*/
```

```
if(typeof(var1) === 'number' && ty
```

```
    // var1 y var2 se suman
```

```
    return var1 + var2;
```

```
}
```

```
/*
```

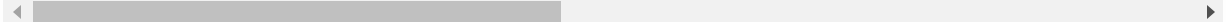
```
    Tipo guard donde var1 y var2 son  
    bloque del condicional
```

```
*/
```

```
if(typeof(var1) === 'string' && tv
```

```
    // var1 y var2 se concatenan
    return var1 + var2;
}

/*
    Para el resto de combinaciones (
    tipo number y, por otro lado, va
    se arroja una excepción
*/
throw new Error('Parámetros inváli
}
```



- El tipo *guard* en objetos permite comprobar la existencia o no de propiedades utilizando la partícula *in*.


```
type PermisosAdministrador = {  
    id: number;  
    servidor: boolean;  
    red: boolean;  
}
```

```
type PermisosProgramador = {  
    id: number;  
    repositorio: boolean;  
    web: boolean;  
}
```

```
type PermisosGeneral = PermisosAdmin
```

```
function mostrarPermisos(permisos: P
```

```
/*
```

```
    Corregido porque la propiedad id
```

```
correcto porque la propiedad la
PermisosAdministrador y Permisos
*/
console.log(permisos.id);

/*
Error porque la propiedad reposi
tipo PermisosProgramador
*/
// console.log(permisos.repositori

/*
Tipo guard para verificar que el
repositorio
*/
if('repositorio' in permisos) {

// Ahora no se produce ningún er
```

```

        // Ahora no se produce ningun error
        console.log(permisos.repositorio);
    }

    // Tipo guard para verificar que e
    if('web' in permisos) {

        console.log(permisos.web);
    }
}

/*
    La consola mostrará:
        1
        true
        true
*/
mostrarPermisos({
    id: 1
});

```

```
    repository: true,  
    web: true  
});
```

- Sin embargo, el uso de *in* como tipo *guard* es poco flexible cuando el número de propiedades es importante y, además, es fácil cometer errores escribiendo el string a la izquierda de *in*.
- Una mejor opción es emplear *instanceof*, pero será necesario utilizar una clase para definir los tipos.

```
class PermisosAdministrador {  
    constructor(  
        public id: number,  
        public servidor: boolean,  
        public red: boolean) { }  
}
```

}

```
class PermisosProgramador {  
    constructor(  
        public id: number,  
        public repositorio: boolean,  
        public web: boolean) { }  
}
```

```
type PermisosGeneral = PermisosAdmin
```

```
function mostrarPermisos(permisos: P
```

```
/*
```

```
    Correcto porque la propiedad id  
    PermisosAdministrador y Permisos
```

```
*/
```

```
----- 1 - 1 - / - - - - - 1 - 1 -
```

```
console.log(permisos.id);

/*
    Error porque la propiedad reposi
    clase PermisosProgramador
*/
// console.log(permisos.repositori

/*
    Tipo guard para verificar que el
    clase PermisosProgramador

*/
if(permisos instanceof PermisosPro
    console.log(permisos.repositorio
    console.log(permisos.web);
}
}
```

```
const permisosProgramador = new Perm
```

```
/*
```

```
    La consola mostrará:
```

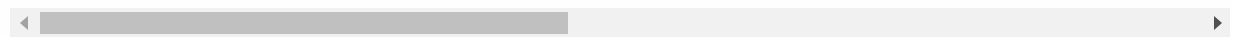
```
        1
```

```
        true
```

```
        true
```

```
*/
```

```
mostrarPermisos(permisosProgramador)
```



- La partícula *instanceof* no puede utilizarse con interfaces.
- Por último, también existe la posibilidad de declarar funciones de tipo *guard*, que son tipadas con la expresión *<variable> is <type>* (aunque

realmente devuelven un booleano en tiempo de ejecución).

```
/*
```

```
Función de tipo guard. Recibe un p  
se trata de un instancia de Permis
```

```
*/
```

```
function esPermisosAdministrador(  
  permisos: any): permisos is Permis  
  
  return permisos instanceof Permiso  
}
```

```
function mostrarPermisos(permisos: P  
  
  console.log(permisos.id);
```



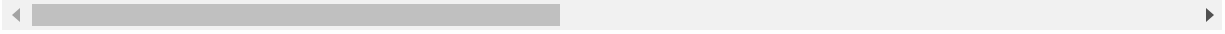
```
/*  
    Invocación a función de tipo gua  
    una instancia de la clase Permis  
*/  
if(esPermisosAdministrador(permiso  
    console.log(permisos.red);  
    console.log(permisos.servidor);  
})  
}
```

```
const permisosProgramador = new Perm
```

```
/*  
  
    La consola mostrará:  
    1  
    true  
    true
```

**/*

```
mostrarPermisos(permisosProgramador)
```



- También se puede aplicar la sobrecarga de funciones para evitar la ambigüedad de tipos en los parámetros de una función.

Tipo *discriminated union*

- Los tipos *guard* con *instanceof* no pueden utilizarse con interfaces porque *instanceof* es una instrucción de JavaScript, lenguaje que no soporta las interfaces.
- Una alternativa es utilizar el concepto de *discriminated union*, que consiste en definir en la

interfaz una propiedad (con nombre arbitrario)
tipado como literal.

```
interface Paloma {
```

```
  /*
```

```
    Los objetos que implementen esta  
    propiedad tipo con el valor a pa
```

```
  */
```

```
  tipo: 'paloma';
```

```
  velocidadAerea: number;
```

```
}
```

```
interface Caballo {
```

```
  /*
```

```
    Los objetos que implementen esta  
    propiedad tipo con el valor a ca
```

```
  */
```

```
  tipo: 'caballo';
```

```
    ,  
    velocidadTerrestre: number;  
}
```

```
type Animal = Paloma | Caballo;
```

```
function moverAnimal(animal: Animal)
```

```
    /*
```

```
        Se discrimina el tipo mediante 1  
        Caballo)
```

```
    */
```

```
switch (animal.tipo) {
```

```
    case 'paloma':
```

```
        console.log(`Velocidad de la p  
        break;
```

```
    case 'caballo':
```

```
        console.log(`Velocidad del cab
```

```
        console.log('Velocidad del car  
break;  
}  
}
```

```
const caballo: Caballo = {
```

```
    /*
```

```
        Es obligatorio que el valor de t  
        la interfaz Caballo
```

```
    */
```

```
    tipo: 'caballo',
```

```
    velocidadTerrestre: 200
```

```
}
```

```
// Velocidad del caballo: 200
```

```
moverAnimal(caballo);
```



- Los *discriminated union* también están disponibles para las clases.

```
class Paloma {  
    constructor(public tipo: 'paloma',  
  
}
```

```
class Caballo {  
    constructor(public tipo: 'caballo'  
}
```



Tipo *casting*

- A veces, determinadas funciones retornan un tipo no esperado.

```
<!--
```

```
    Código HTML
```

```
-->
```

```
<input type="text" id="input" />
```

```
/*
```

```
    Código de TypeScript
```

```
*/
```

```
// El método getElementById devuelve
```

```
const inputElemento = document.getEl
```

```
// Correcto porque la propiedad inne
```

```
inputElemento.innerHTML= 'valor';
```

```
/*
```

```
Error porque la propiedad value no  
embargo, el objeto realmente sí ti  
inputElemento hace referencia al e  
propiedad
```

```
*/
```

```
// inputElemento.value= 'a';
```



- Se puede forzar a un tipo en particular mediante un mecanismo conocido como *casting*, utilizando la sintaxis `<>` (menos recomendable) o la partícula *as* (más recomendable).

```
// Se realiza un casting al tipo HTM
```

```
const inputElemento1 = <HTMLInputEle
```


```
// Correcto porque la propiedad valu
```

```
inputElemento1.value = 'valor';
```



```
// Se realiza un casting al tipo HTML  
const inputElemento2 = document.getE
```

```
// Correcto porque la propiedad value  
inputElemento2.value = 'valor';
```



- Es compromiso del programador garantizar que el casting realizado es correcto, dado que TypeScript no realiza ninguna comprobación sobre los datos y pueden producirse errores en tiempo de ejecución si el tipo no es el esperado.

Tipos índice

- En ocasiones es necesario que los nombres de las propiedades de los objetos sean dinámicos.

```
const errorPropiedades = {  
  usuario: 'El usuario debe contener  
  password: 'La contraseña debe esta  
  /* ... */  
}
```

- TypeScript proporciona los tipos índice mediante la notación corchete para indicar el tipo del nombre de las propiedades, pero no sus nombres.

```
/*  
  
  Todos los objetos que implementen  
  propiedades y valores de tipo stri  
  puede utilizarse cualquier otro)  
*/
```

```
interface ErrorContenedor {
```

```
[propiedadNombre: string]: string;

}

/*
    Las propiedades y los valores de
    todos de tipo string
*/
const errorPropiedades1: ErrorConten
  usuario: "El usuario debe contener
  password: "La contraseña debe esta
}

/*
    También correcto porque en este caso
    directamente a tipo string
*/
const errorPropiedades2: ErrorConten
```

```
123456: "Esto es un mensaje de pru  
}  
  
/*  
    Error porque el objeto que impleme  
    contener propiedades cuyo nombre e  
*/  
/*  
const errorPropiedades3: ErrorConten  
    [true]: "Esto es un mensaje de pru  
}  
*/
```



- No se pueden agregar varios índices de tipo a una interfaz, pero sí se pueden añadir otras

propiedades normales, siempre y cuando cumplan con el índice de tipo.

```
interface ErrorContenedor {  
    [property: string]: string;
```

```
    /*
```

```
        Correcto porque el nombre de la  
        también lo es
```

```
    */
```

```
    id: string;
```

```
    /*
```

```
        Error porque la propiedad es de  
        y esto está en contradicción con
```

```
    */
```

```
    errores: number;
```

```
    }
```

ۛ



Genéricos

- Introducción
- Creación de un tipo genérico
- Restricciones en los genéricos
- La partícula *keyof*
- Tipos genéricos en clases
- Tipos genéricos en interfaces
- Tipo *Partial*
- Tipo *Readonly*

Introducción

- Algunos tipos de datos vistos con anterioridad utilizan tipos genéricos.
- Un tipo genérico es un tipo indefinido que toma un tipo específico en tiempo de ejecución.
- Por ejemplo, los arrays utilizan un tipo genérico porque almacenan elementos cuyos valores son, a priori, indefinidos. El tipo específico del genérico para el array es tomado en tiempo de ejecución a partir de lo establecido con el operador diamante `<>` o, a veces, puede ser inferido directamente por TypeScript.

*/**

Error porque Array utiliza un tipo que indique qué tipo de elementos inferir automáticamente en este ca

**/*

--

--


```
// const array1: Array = [];
```

```
/*
```

```
    Correcto porque se ha establecido  
    diamante) como el tipo específico
```

```
*/
```

```
const array2: Array<string> = [];
```

```
// Otra forma de declarar un array
```

```
const array3: string[] = [];
```

```
// TypeScript infiere automáticamente
```

```
const array4 = ['fútbol', 'baloncest
```



- Otro tipo importante que utiliza un genérico es la promesa. En este caso no es obligatorio indicar el

tipo específico que debe utilizar la promesa porque se considera por defecto que es *unknown*.

// Creación de la promesa

```
const promesa1 = new Promise((resolve  
  setTimeout(() => {
```

```
    // La promesa resuelve con un va  
    resolve('Promesa resuelta');
```

```
  }, 2000);  
});
```

// Uso de la promesa

```
promesa1.then((resultado) => {
```

```
  /*
```

```
    La variable resultado es de tipo
```

```

    La variable resultado es de tipo
    de tipo unknown porque en la dec
    el tipo específico a utilizar
*/
console.log(resultado);

// Error porque la propiedad lengt
// console.log(resultado.length);
});

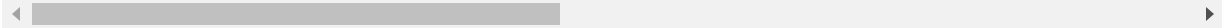
/*
    La promesa se declara ahora establ
    (string) mediante el operador diam
*/
const promesa2 = new Promise<string>
    setTimeout(() => {
        resolve('Promesa resuelta');
    }, 2000);

```

```
    }, 2000),  
  });
```

```
// La promesa se resuelve correctamente  
promesa2.then((resultado) => {
```

```
    // TypeScript puede ahora inferir  
    console.log(resultado.length);  
});
```



Creación de un tipo genérico

- Sin los genéricos, TypeScript no puede inferir las propiedades de los objetos cuando son pasados por parámetros a las funciones.

```
/*
```

```
Esta función recibe dos objetos y
```

```
    Esta función recibe dos objetos y  
    infiere que el valor retornado es  
    establecerlo) porque el método assign  
*/
```

```
function unir(obj1: object, obj2: ob  
    return Object.assign(obj1, obj2);  
}
```

```
const objeto = unir(  
    { numero1: 1 },  
    { numero2: 2 }  
);
```

```
/*
```

```
    Error porque objeto es de tipo obj  
    numero1, ni tampoco la propiedad n
```

```
*/
```

```
// console.log(objeto.numero1);
```

```
// console.log(objeto.numero1);  
// console.log(objeto.numero2);
```

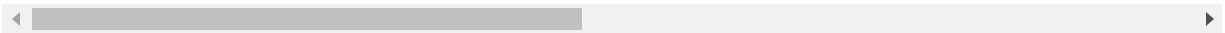
```
/*
```

*En tiempo de ejecución la variable
y numero2. La consola mostrará:*

{ numero1: 1, numero2: 2 }

```
*/
```

```
console.log(objeto);
```



- Para crear un genérico habitualmente se suele utilizar la notación T (y sus sucesivas letras). Posteriormente en tiempo de ejecución se establece qué tipo se utiliza mediante el operador diamante <> o directamente TypeScript puede inferirlo automáticamente.

```
/*
```

T y U son genéricos que tomarán ti

```
TypeScript entiende el funcionamie  
se retorna la intersección  $T \& U$  (  
*/
```

```
function unir<T, U>(obj1: T, obj2: U  
    return Object.assign(obj1, obj2);  
}
```

```
/*  
    Para este caso el tipo genérico  $T$   
    es { numero2: number }. TypeScript  
*/
```

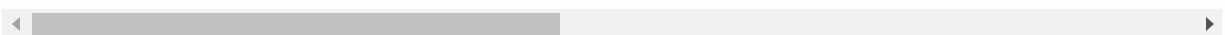
```
const objeto1 = unir(  
    { numero1: 1 },  
    { numero2: 2 }  
);
```

```
/*
```

```
Para objeto1, el tipo retornado po  
{ numero1: number } & { numero2:  
Por tanto, el tipo resultante es:  
{ numero1: number; numero2: numb  
*/  
console.log(objeto1.numero1);
```

```
/*  
En este caso se especifican explíc  
redundante porque TypeScript puede  
expuso anteriormente)  
*/
```

```
const objeto2 = unir<{ numero1: numb  
  { numero1: 1 },  
  { numero2: 2 }  
);
```



Restricciones en los genéricos

- A veces se desea restringir el tipo genérico para que no acepte cualquier tipo posible.

```
function unir<T, U>(obj1: T, obj2: U)  
    return Object.assign(obj1, obj2);  
}
```

```
/*
```

Como el tipo U es genérico, el valor ideal puesto que la función unir u objetos (el resto de tipos los ign

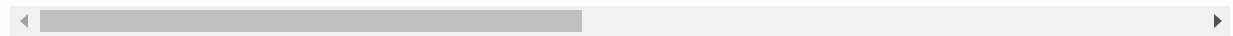
```
*/
```

```
const objeto = unir({ numero: 1 }, 3
```

```
// { numero: 1 }
```

```
console.log(objeto);
```

```
console.log(objeto),
```



- Para restringir un tipo genérico se utiliza la partícula *extends*, seguido del tipo que se quiere restringir.

```
/*
```

```
    Los tipos genéricos T y U heredan  
    objetos de forma obligatoria (con
```

```
*/
```

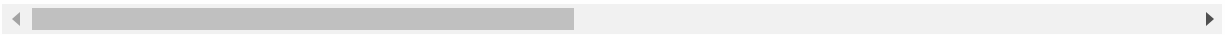
```
function unir<T extends object, U ex  
    return Object.assign(obj1, obj2);  
}
```

```
/*
```

```
    Error porque el tipo genérico U de  
    propiedades) y no un valor de tipo
```

```
*/
```

```
// const objeto = unir({ numero: 1 }
```



- Las restricciones sobre los tipos genéricos pueden incluir cualquier otro tipo, incluyendo los tipo *union*, las clases y las interfaces.
- Con las restricciones de interfaces se puede forzar a que un tipo genérico posea obligatoriamente alguna propiedad o propiedades establecidas en una interfaz.

```
interface TienePropiedadLength {  
    length: number;  
}
```

```
const describir = <T extends TienePr
```

```
    let descripcion: string;  
    if(valor.length === 0) {
```

```
    descripcion = 'No contiene eleme
} else if(valor.length === 1) {
    descripcion = 'Contiene 1 elemen
} else {
    descripcion = `Contiene ${valor.
}

return [valor, descripcion];
}

/*
    Correcto porque el tipo Array pose
    [ [ 1, 2, 3 ], 'Contiene 3 eleme
*/
console.log(describir([1, 2, 3]));

/*
```

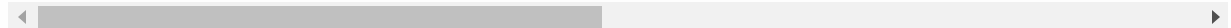
```
Correcto porque el tipo string pos  
[ 'hola', 'Contiene 4 elementos'  
*/  
console.log(describir('hola'));
```

```
// Error porque el tipo number no po  
// console.log(describir(5));
```

La partícula *keyof*

- A veces no se puede garantizar dentro de una función que una determinada propiedad exista para un objeto concreto. Esto genera un error en TypeScript.

```
const extraer = <T extends object, U  
/*
```


```
Error porque no se puede garanti  
nombre es el valor de la variabl  
*/  
// return objeto[prop];  
}  

```

- La partícula *keyof* se utiliza para establecer que un tipo genérico debe ser el nombre de propiedad de otro tipo genérico.

```
/*  
T hereda de objeto y U es un tipo  
las propiedades de T  
*/  
const extraer = <T extends object, U  
  return objeto[propiedad];  
}
```

```
// Correcto porque la propiedad numero  
extraer({ numero: 1 }, 'numero');
```

```
// Error porque la propiedad edad no  
// extraer({ numero: 1 }, 'edad');
```



Tipos genéricos en clases

- Las clases en TypeScript también pueden utilizar tipos genéricos.

```
class Resultado<T> {
```

```
// el atributo error es tipado con  
constructor(public esCorrecto: boo
```

```
        console.log(typeof(error))
    }
}
```

```
/*
```

El tipo genérico T de la clase es indicarlo porque TypeScript puede 'error: 42' es un valor de tipo st

```
*/
```

```
const error1 = new Resultado<string>
```

```
/*
```

El tipo genérico T de la clase es indicarlo porque TypeScript puede valor de tipo number

```
*/
```

```
const error2 = new Resultado<number>
```



```
/*
```

```
El tipo genérico T es object porqu  
la función typeof de JavaScript
```

```
*/
```

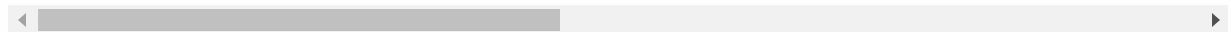
```
const error3 = new Resultado<object>
```

```
/*
```

```
Error porque se indica que el tipo  
parámetro del constructor se le pa
```

```
*/
```

```
// const error4 = new Resultado<stri
```



- Los métodos definidos en las clases también pueden utilizar tipos genéricos.

```
/*
```

```
Se define una clase que utiliza un  
booleano, número y string en tiempo
```

```
    boolean, number y string en tiempo  
    */
```

```
class Almacen<T extends boolean | nu
```

```
    private data: T[] = [];
```

```
    /*
```

```
        Método que utiliza un tipo T def  
        permite agregar elementos en el
```

```
    */
```

```
    public agregarElemento(elemento: T  
        this.data.push(elemento);
```

```
    }
```

```
    // Método para eliminar elementos
```

```
    public eliminarElemento(elemento:  
        if(this.data.indexOf(elemento) =
```

```

        return;
    }

    this.data.splice(this.data.index
}

public obtenerElementos() {
    return this.data;
}

}

/*
    El objeto almacen1 almacenará valo
    la clase Almacen
*/

const almacen1: Almacen<string> = ne
almacen1.agregarElemento('tenis');
- - - - -

```

```
almacen1.agregarElemento('baloncesto')
almacen1.agregarElemento('fútbol');
```

```
// [ 'tenis', 'baloncesto', 'fútbol' ]
console.log(almacen1.obtenerElemento
```

```
almacen1.eliminarElemento('tenis');
```

```
// [ 'baloncesto', 'fútbol' ]
```

```
console.log(almacen1.obtenerElemento
```

```
/*
```


```
    El objeto almacen2 almacenará valo  
    la clase Almacen
```

```
*/
```

```
const almacen2: Almacen<number> = ne  
almacen2.agregarElemento(1);
```

```
- - - - -
```

```
almacen2.agregarElemento(29);  
almacen2.agregarElemento(2);  
  
// [ 1, 29, 2 ]  
console.log(almacen2.obtenerElemento  
  
almacen2.eliminarElemento(2);  
  
// [ 1, 29 ]  
console.log(almacen2.obtenerElemento
```



Tipos genéricos en interfaces

- Los tipos genéricos funcionan de forma similar en las interfaces.

// La interfaz utiliza el genérico T

```
interface EstadoInterfaz<T> {  
    guardar(evento: T): void;  
    obtenerTodos(): T[];  
}
```

*/**

*La clase Estado utiliza el genérico
EstadoInterfaz, que a su vez tambi*

**/*

```
class Estado<T> implements EstadoInt
```

```
    private lista: T[] = [];
```

```
    guardar(evento: T): void {  
        this.lista.push(evento);  
    }
```

```
    obtenerTodos(): T[] {  
        return this.lista;  
    }  
}
```

// La interfaz EventoInterfaz utiliz

```
interface EventoInterfaz<T> {  
    codigo: T;  
}
```

*/**

Utiliza el tipo number para establ

*la interfaz EventoInterfaz y segui
específico del genérico para la cl*

**/*

```
const estado = new Estado<EventoInte
```

// Correcto

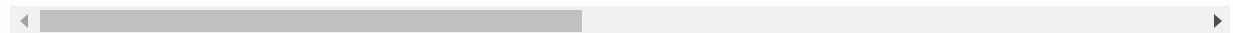
```
estado.guardar({ codigo: 200 });  
estado.guardar({ codigo: 500 });
```

// Se muestran por consola todos los

```
estado.obtenerTodos().forEach(event
```

// Error porque el valor de la propi

// estado.guardar({ codigo: '500' })



- También se pueden declarar interfaces con múltiples tipos genéricos.

```
interface EjecutableInterfaz<T, U> {  
    ejecutar(input: T): U;  
}
```



Tipo *Partial*

- A veces puede resultar interesante crear un objeto vacío en TypeScript y posteriormente añadir sus propiedades de forma dinámica a partir de una interfaz o clase.

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

```
/*
```

```
    Error porque el objeto retornado (  
    declaradas en la interfaz Interfaz
```

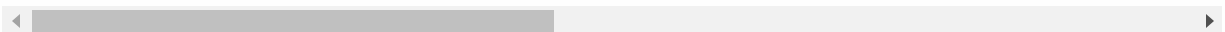
```
*/
```

```
/*
```

```
,  
  
const crearPersona1 = (): InterfazPe  
    return {};  
}  
*/
```

```
/*  
    Correcto porque el objeto retornad  
    InterfazPersona  
*/
```

```
const crearPersona2 = (): InterfazPe  
  
    return {  
        nombre: 'Marcos',  
        apellidos: 'Rodríguez',  
        edad: 25  
    };  
};
```



- Para poder generar propiedades tipadas de forma dinámica se puede declarar un objeto de tipo *Partial*.
- *Partial* es un tipo que utiliza genéricos para establecer las propiedades que posteriormente se irán añadiendo dinámicamente al objeto.
Básicamente convierte a todas las propiedades en opcionales.

```
const crearPersona = (): InterfazPer
```

```
/*
```

```
    El objeto persona es de tipo Par  
    en este caso
```

```
*/
```

```
const persona: Partial<InterfazPer
```

```
/*
```

Ahora pueden agregarse las propiedades dinámicas

```
*/
```

```
persona.nombre = 'Marcos';
```

```
persona.apellidos = 'Rodríguez';
```

```
persona.edad = 25;
```

```
// Casting al tipo InterfazPersona
```

```
return persona as InterfazPersona;
```

```
}
```

```
const persona = crearPersona();
```



Tipo *Readonly*

- El genérico *Readonly* es un tipo que utiliza genéricos para crear variables cuyo valor no puede ser modificado.

```
// El genérico para Readonly es Arra  
const array: Readonly<Array<string>>
```

```
/*  
Error porque no se pueden agregar/  
Readonly  
*/
```

```
// array.push('baloncesto');  
// array.pop();
```

```
interface ObjetoInterfaz {  
    nombre: string;  
    apellidos: string;  
}
```

```
}
```

```
const objeto: Readonly<ObjetoInterfa  
  nombre: 'Marcos',  
  apellidos: 'Rodríguez'  
}
```

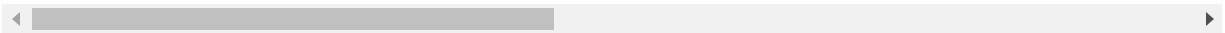
```
/*
```

```
  Error porque no se pueden agregar/  
  Readonly
```

```
*/
```

```
// objeto.edad = 23;
```

```
// delete objeto.apellidos;
```



Decoradores

- Introducción
- Decoradores de clases
- Decoradores de atributos
- Decoradores de accesores
- Decoradores de métodos
- Decoradores de parámetros en métodos
- Orden de ejecución de los decoradores
- Cambiando una clase con un decorador
- Ejemplo de validación de los atributos de una clase con decoradores
- Ejemplo de *autobind* con decoradores

Introducción

- Un decorador es un tipo especial de declaración utilizado en [meta-programación](#) que acompaña a un elemento declarado (clase, objeto, método o atributo).
- Los decoradores utilizan la sintaxis *@nombre_decorador* encima del elemento al que acompañan o decoran, donde *nombre_decorador* es una función especial que el programador deberá implementar y que será invocada en tiempo de ejecución para obtener información y modificar el comportamiento del elemento decorado.
- Los parámetros que recibirá la función decoradora dependerá del tipo de elemento decorado.

- Los decoradores son una funcionalidad experimental y es necesario habilitar esta funcionalidad explícitamente en el archivo de configuración *tsconfig.json*.

```
{  
  "compilerOptions": {  
    /* ... */  
    "experimentalDecorators": true  
  }  
}
```

Decoradores de clases

- Un decorador de clase se ubica encima de la declaración de una clase y puede utilizarse para observar, modificar o reemplazar su comportamiento.
- Las funciones decoradoras únicamente son invocadas una vez (independientemente de que se creen o no instancias de la clase).
- Las funciones decoradoras de clase reciben un único parámetro (generalmente denominado *target* y tipado como *Function*), que hace referencia a la clase decorada.

```
// Función que posteriormente se uti
function Logger(target: Function) {

    console.log('Decorador ejecutado')
```

```
/*
```

```
En este caso target hace referen  
atributos y métodos estáticos y  
estáticos
```

```
*/
```

```
console.log(target);
```

```
/*
```

```
Itera todos los atributos estáti  
estáticos
```

```
*/
```

```
for(let nombreAtributo in target)
```

```
// nombre: Alejandro
```

```
console.log(`${nombreAtributo}:`
```

```
// El decorador cambia el valor
```

```
    if(nombreAtributo === 'nombre')  
        target[nombreAtributo] = 'Juan'  
    }  
}  
}
```

```
/*
```

```
    Uso del decorador en la clase Pers  
    incluso aunque no se cree una inst
```

```
*/
```

```
@Logger
```

```
class Persona {
```

```
    // Atributo público estático
```

```
    static nombre: string = 'Alejandro'
```

```
    // Método público estático
```

```
    static mostrarNombre() {
```

```
        console.log(Persona.nombre);  
    }  
}
```

// Juan

```
console.log(Persona.nombre);
```



- También se puede decorar una clase con varios decoradores. El primer decorador que se ejecutará será aquel más cercano a la declaración de la clase y así sucesivamente hasta llegar al más lejano.
- Un decorador también puede recibir parámetros de configuración. Para ello puede crearse una función que reciba los parámetros de

configuración y retorne un decorador, es decir, que retorne una función con el parámetro *target*.

```
// Función que devuelve el decorador
```

```
function Plantilla(id: string): Func
```

```
// Este console.log será el primer  
console.log('Función que retorna e
```

```
// Decorador Plantilla
```

```
return (target: Function) => {
```

```
// Este console.log será el terc  
console.log(`Decorador Plantilla  
}  
}
```

```
// Decorador Logger
```

```
function Logger(target: Function) {  
  
    // Este console.log será el segund  
    console.log('Decorador Logger ejec  
}  
  
/*  
    El decorador Logger se ejecutará a  
    de la invocación a la función Plan  
*/  
@Plantilla('app')  
  
@Logger  
class Persona {  
    constructor() { }  
}  
  
/*
```

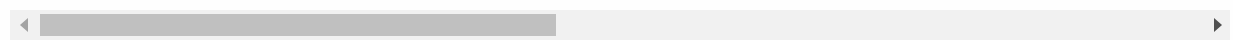
La consola mostraría:

Función que retorna el decorador

Decorador Logger ejecutado

Decorador Plantilla ejecutado co

** /*



Decoradores de atributos

- Un decorador de atributo se coloca justo antes de la declaración de un atributo.
- Al igual que en los decoradores de clase, los decoradores de atributo se ejecutan incluso si no se crea un objeto de la clase donde se ubican.
- Un decorador de un atributo recibe dos parámetros:

- El *target* del atributo, cuyo valor depende de:
 - Si el atributo no es estático y el valor de la propiedad *target* del archivo *tsconfig.json* está establecido a ECMAScript 5 (*es5*) o inferior: el *target* es el prototipo (los métodos que hereda un objeto) de un objeto creado a partir de la clase donde se encuentra el atributo decorado.
 - Si el atributo no es estático y el valor de la propiedad *target* del archivo *tsconfig.json* está establecido a ECMAScript 6 (*es6*) o superior: el *target* es un objeto vacío.
 - Si el atributo es estático: el *target* es el constructor de la clase.
- El nombre del atributo.

function LogAtributoNoEstatico(targe

```
/*
```

```
El valor mostrado depende del va  
tsconfig.json. La consola muestra  
{constructor: f, mostrarNombre  
{}} (para ECMAScript 6 o super
```

```
*/
```

```
console.log(target);
```

```
// nombre
```

```
console.log(name);
```

```
}
```

```
function LogAtributoEstatico(target:
```

```
// [Function: Persona] (construct  
console.log(target);
```

```
// ...
```

```
// edad  
console.log(name);  
}
```

```
class Persona {
```

```
// Se decora el atributo público n  
@LogAtributoNoEstatico  
nombre: string;
```

```
// Se decora el atributo público e  
@LogAtributoEstatico  
static edad: string;
```

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

```
mostrarNombre() {  
    console.log(this.nombre);  
}  
}
```

```
/*
```

*Prototipo de un objeto de la clase
primer parámetro en un decorador d*

*propiedad target es ECMAScript 5 o
tsconfig.json). La consola mostrar
{constructor: f, mostrarNombre:*

```
*/
```

```
console.log(Object.getPrototypeOf(ne
```



Decoradores de accesores

- Un decorador de un accesor (método *get* o *set*) recibe tres parámetros:
 - El *target* del accesor, cuyo valor depende de:
 - Si el accesor no es estático y el valor de la propiedad *target* del archivo *tsconfig.json* está establecido a ECMAScript 5 (*es5*) o inferior: el *target* es el prototipo (los métodos que hereda un objeto) de un objeto creado a partir de la clase donde se encuentra el accesor decorado.
 - Si el accesor no es estático y el valor de la propiedad *target* del archivo *tsconfig.json* está establecido a ECMAScript 6 (*es6*) o superior: el *target* es un objeto vacío.
 - Si el accesor es estático: el *target* es el constructor de la clase.
 - El nombre del accesor.

- El descriptor, que es un objeto con información sobre el accesor y que se encuentra tipado con *PropertyDescriptor*. Entre las propiedades del descriptor se encuentran:
 - *get*: accesor de obtención de la propiedad del accesor decorado, o undefined si no existe.
 - *set*: accesor de modificación de la propiedad del accesor decorado, o undefined si no existe.
 - *enumerable*: booleano que indica si el accesor aparece durante la enumeración de las propiedades del objeto.
 - *configurable*: booleano que indica si el accesor puede ser eliminado y si pueden modificarse los atributos del descriptor.

```
function Log(target: any, name: stri
```

```
/*
```

```
El valor mostrado depende del va  
tsconfig.json. La consola muestra  
{constructor: f, mostrarNombre  
{}} (para ECMAScript 6 o super
```

```
*/
```

```
console.log(target);
```

```
// nombre
```

```
console.log(name);
```

```
/*
```

```
{
```

```
get: undefined,
```

```
set: [Function: set],
```

```
        enumerable: false,
        configurable: true
    }
    */
    console.log(descriptor);
}

class Persona {

    private _nombre: string;

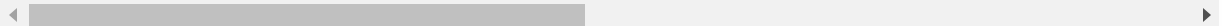
    constructor(nombre: string) {
        this._nombre = nombre;
    }

    // Se decora el accesor nombre

    @Log
```



```
set nombre(nombre: string) {  
    this._nombre = nombre;  
}  
  
mostrarNombre() {  
    console.log(`${this.nombre}`);  
}  
}
```



- No se puede decorar dos accesores que tienen el mismo nombre (un método *get* y un método *set* con el mismo nombre).

Decoradores de métodos

- Un decorador de un método recibe los mismos parámetros que un accesor:

- El *target* del método, cuyo valor depende de:
 - Si el método no es estático y el valor de la propiedad *target* del archivo *tsconfig.json* está establecido a ECMAScript 5 (*es5*) o inferior: el *target* es el prototipo (los métodos que hereda un objeto) de un objeto creado a partir de la clase donde se encuentra el método decorado.
 - Si el método no es estático y el valor de la propiedad *target* del archivo *tsconfig.json* está establecido a ECMAScript 6 (*es6*) o superior: el *target* es un objeto vacío.
 - Si el método es estático: el *target* es el constructor de la clase.
- El nombre del método.
- El descriptor, que es un objeto con información sobre el método. Entre las

propiedades se encuentran:

- *value*: método decorado.
- *writable*: booleano que indica si el método puede ser modificado.
- *enumerable*: igual que en los decoradores de accesorios.
- *configurable*: igual que en los decoradores de accesorios.

```
function Log(target: any, name: stri
```

```
// {constructor: f, mostrarNombre:  
console.log(target);
```

```
// mostrarNombre  
console.log(name);
```

```
/*
```

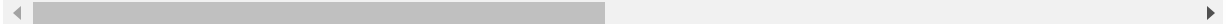
```
{
    value: f,
    writable: true,
    enumerable: true,
    configurable: true
}
*/
console.log(descriptor);
}
```

```
class Persona {

    private nombre: string;

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

```
// Se decora el método mostrarNomb  
@Log  
mostrarNombre() {  
    console.log(`${this.nombre}`);  
  
}  
}
```



Decoradores de parámetros en métodos

- Un decorador de un parámetro en un método recibe tres argumentos:
 - El *target* del parámetro de un método, cuyo valor depende de:

- Si el valor de la propiedad *target* del archivo *tsconfig.json* está establecido a ECMAScript 5 (*es5*) o inferior: el *target* es el prototipo (los métodos que hereda un objeto) de un objeto creado a partir de la clase donde se encuentra el parámetro.
- Si el valor de la propiedad *target* del archivo *tsconfig.json* está establecido a ECMAScript 6 (*es6*) o superior: el *target* es un objeto vacío.
- El nombre del método donde se encuentra el parámetro.
- La posición del parámetro, donde 0 es la posición del primer parámetro.

```
function Log(target: any, name: stri
```

```
// {constructor: f, mostrarNombre:
```

```
console.log(target);

// mostrarNombre
console.log(name);

// 1
console.log(position);
}
```

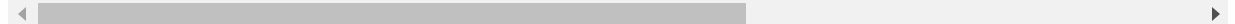
```
class Persona {

    private nombre: string;

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

    .....
```

```
// Se decora el parámetro apellido  
mostrarNombre(nombre: string, @Log  
    console.log(`${nombre}`);  
}  
}
```



Orden de ejecución de los decoradores

- Independientemente de las instancias que se creen de la clase, los decoradores únicamente se ejecutan una vez.
- El orden de ejecución de los decoradores en una clase se mantiene con respecto al orden de aparición, excepto en el caso de los decoradores de los parámetros, que son ejecutados antes que

los decoradores de los métodos en los que se encuentran.

```
function LogParametro(target: any, n  
    // Tercer console.log que se ejecu  
    console.log('Decorador del parámet  
}
```

```
function LogAtributo(target: any, na  
    // Primer console.log que se ejecu  
    console.log('Decorador del atribut  
}
```

```
function LogMetodo(target: any, name  
    // Cuarto console.log que se ejecu  
    console.log('Decorador del método'
```

```
}
```

```
function LogAccesor(target: any, nam  
    // Segundo console.log que se ejec  
    console.log('Decorador del accesor  
)
```

```
class Persona {
```

```
    @LogAtributo
```

```
    private _nombre: string;
```

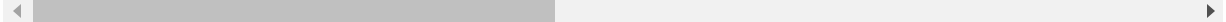
```
    constructor(nombre: string) {
```

```
        this._nombre = nombre;
```

```
    }
```

```
    @LogAccesor
```

```
set nombre(nombre: string) {  
    this._nombre = nombre;  
}  
  
@LogMetodo  
mostrarNombre(@LogParametro nombre  
    console.log(`${nombre}`);  
}  
}
```



Cambiando una clase con un decorador

- Un decorador puede modificar el constructor de una clase para añadirle alguna funcionalidad adicional.

- La complejidad reside en el genérico (T), que debe ser adaptado para tipar a cualquier clase e incluirse en la devolución de la función decoradora. Este tipo genérico T hereda de un objeto especial con una única propiedad: *new (...args: any[])*, que tiene las siguientes características:
 - Indica que el tipo genérico T necesita establecer obligatoriamente un constructor en su definición con un número indefinido de parámetros de cualquier tipo.
 - Debe retornar un objeto que incluya todos los atributos tipados de la clase decorada.
- El *target* de la función decoradora será del tipo del genérico T.

```
function Logger( ) {
```

```
console.log('Fábrica de un decorad
```

```
/*
```

```
El genérico T hereda de un objet  
new (...args: any[]), que hace r  
un objeto que incluya todos los  
{ nombre: string }
```

```
*/
```

```
return <T extends { new (...args:
```

```
console.log('Decorador');
```

```
/*
```

```
Se retorna una nueva clase que  
con el genérico T
```

```
*/
```

```
return class extends Target {
```

```
return class extends Target {
```

```
    /*
```

```
        El constructor de la clase h  
        obligatoriamente un número i
```

```
    */
```

```
    constructor(...args: any[]) {
```

```
        /*
```

```
            Invocación al constructor  
            parámetros que se pasen en
```

```
        */
```

```
        super(...args);
```

```
        console.log('Constructor nue  
        this.nombre = 'Carlos';
```

```
    }
```

```
}
```

```
    }  
  }  
}
```

```
// Se decora la clase Persona con el  
@Logger()  
class Persona {  
    constructor(public nombre: string)  
        console.log('Constructor origina  
    }  
}
```

```
const persona = new Persona('Marta')
```

```
/*
```

```
El decorador modifica el valor del  
es Marta. La consola mostrará:
```

```
Constructor
```

carlos

**/*

```
console.log(persona.nombre);
```



- Se puede modificar el comportamiento de métodos y accedores, pero no de atributos y parámetros, dado que TypeScript ignora las devoluciones de las funciones decoradoras.
- También se pueden alterar los descriptores para accedores y métodos.

Ejemplo de validación de los atributos de una clase con decoradores

- Un ejemplo de decorador es el uso de validadores para los atributos de una clase. A continuación, se presenta un ejemplo de implementación y aplicación de tres decoradores de atributos:
 - *@Requerido*: los atributos decorados con este decorador deberán contener un valor (no pueden ser *null* o *undefined*).
 - *@Min*: los atributos decorados con este decorador tienen que ser números y no pueden ser menores que el valor pasado al decorador.
 - *@Max*: los atributos decorados con este decorador tienen que ser números y no pueden ser mayores que el valor pasado al decorador.

/*

*Se implementa una clase (que imple
validación (Requerido, Min y Max)*

*/

```
interface Validador {  
    validar(valor: any): boolean;  
    msg(nombrePropiedad: string): string;  
}
```

```
class ErrorValidacionRequerido implements Validador {  
  
    public validar(valor: any): boolean {  
        return (valor) ? true : false;  
    }  
  
    public msg(nombrePropiedad: string): string {  
        return `El atributo ${nombrePropiedad} es requerido`;  
    }  
}
```

```
class ErrorValidacionMin implements
```

```
    constructor(private valorMin: number)
```

```
    public validar(valor: number): boolean
```

```
        return (valor < this.valorMin) ?  
    }
```

```
    public msg(nombrePropiedad: string)
```

```
        return `El atributo ${nombrePropiedad} no cumple con el requisito de ser mayor a ${valorMin}`  
    }  
}
```

```
class ErrorValidacionMax implements
```

```
    constructor(private valorMax: number)
```

```

    public validador(valor: number): boolean {
        return (valor > this.valorMax) ? true : false;
    }

    public msg(nombrePropiedad: string): string {
        return `El atributo ${nombrePropiedad} no cumple con los requisitos.`;
    }
}

```

/*

A continuación, se crea una estructura que almacena todos los atributos que se validan. Esta estructura de datos realmente es una clase, ErroresValidacionClase, que a su vez implementa la interfaz ErroresValidacion. En primer lugar, se define la interfaz ErroresValidacion y, seguidamente,

**/*

```
/*
```

```
El atributo decorado será una prop  
lista de los objetos validadores (  
con múltiples decoradores)
```

```
*/
```

```
interface ErroresValidacionInterfaz
```

```
[property: string]: (  
    ErrorValidacionRequerido |  
    ErrorValidacionMin |  
    ErrorValidacionMax) []
```

```
}
```

```
/*
```

```
Cada atributo en esta clase será e  
  
un atributo decorado. El valor ser  
anteriormente creada (ErroresValid
```

**/*

```
class ErroresValidacionClase {  
    [property: string]: ErroresValidac  
}
```

// Inicialización del objeto de erro

```
const erroresValidacion = new Errore
```

*/**

*Función para inicializar el objeto
array que almacena los objetos val
creado (erroresValidacion)*

**/*

```
function reset(nombreClase: string,
```

```
    if(!(nombreClase in erroresValidac  
        erroresValidacion[nombreClase] =
```

```

    }

    if(!(nombreAtributo in erroresValidacion[nombreClase][n
    }
}

```

```

/*
    A continuación, se implementan las
*/

```

```

// Decorador Requerido

```

```

function Requerido(target: any, nomb
    console.log('Decorador Requerido')

```

```

// target.constructor.name es el n
    reset(target.constructor.name, nom

```

```
erroresValidacion[target.construct  
    new ErrorValidacionRequerido());  
}
```

```
// Decorador Min
```

```
function Min(valor: number) {  
    return (target: any, nombreAtribut  
        console.log('Decorador Min');  
        reset(target.constructor.name, n  
        erroresValidacion[target.constru  
            new ErrorValidacionMin(valor))  
    }  
}
```

```
// Decorador Max
```

```
function Max(valor: number) {
```



```

        return (target: any, nombreAtribut
        console.log('Decorador Max');
        reset(target.constructor.name, n
        erroresValidacion[target.constru
        new ErrorValidacionMax(valor))
    }
}

```

```

// Valida todos los atributos del ob
function validate(objeto: any, mostr

```

```

/*

```

```

    Se comprueba que la clase del ob
    de erroresValidacion

```

```

*/

```

```

if(objeto.constructor.name in erro

```

```

// Se recorren todos los atribut
for (let property of Object.keys

// Se recorren todos los objet
for (let validation of errores

// Se efectúa la validación
if(!validation.validador(obj

// Se muestra el error de
const msg = validation.msg
if(mostrarError) validatio
if(arrojarError) throw new
    }
  }
}
}

```

```
}
```

```
/*
```

```
Por último, se utilizan los decora
```

```
*/
```

```
class Persona {
```

```
/*
```

```
Decorador que fuerza a que el at  
valor vacío
```

```
*/
```

```
@Requerido
```

```
nombre: string;
```

```
/*
```

```
Decoradores que fuerzan que el a
```

```

    0 (mínimo) y 120 (máximo)
*/
@Min(0)
@Max(120)
edad: number;

constructor(nombre: string, edad:
    this.nombre = nombre;
    this.edad = edad;
}
}

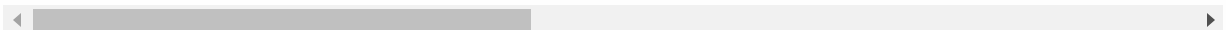
const persona1 = new Persona('Marcos
const persona2 = new Persona('Ana',

/*
    Se realiza la validación sin mostr

```

```
    parámetro), pero sí arrojando una
    (tercer parámetro)
*/
validate(persona1, false, true);

// Arroja una excepción porque la ed
// validate(persona2, false, true);
```



- Una librería muy interesante para validar con TypeScript mediante decoradores es [class-validator](#)

Ejemplo de *autobind* con decoradores

- El uso de *this* en JavaScript/TypeScript puede dar lugar a confusión en determinados contextos.

```
<!--
```

```
    Código HTML
```

```
---->
```

```
<button id="boton">Pulsa aquí</button>
```



```
/*
```

```
    Código de TypeScript
```

```
*/
```

```
class Boton {
```

```
    private mensaje = 'Botón pulsado';
```

```
    mostrarMensaje() {
```

```
    /*  
        Referencia al elemento HTMLBut  
        método addEventListener  
  
    */  
    console.log(this);  
  
    /*  
        Muestra undefined (ver más aba  
        addEventListener)  
    */  
    console.log(this.mensaje);  
}  
}
```

```
const boton = new Boton();
```

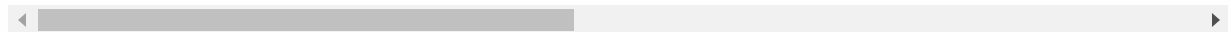
```
const buttonElement = document.getEl
```

```
/*
```

```
Dentro del método mostrarMensaje,  
HTMLElement, en lugar de al
```

```
*/
```

```
buttonElement.addEventListener('click
```



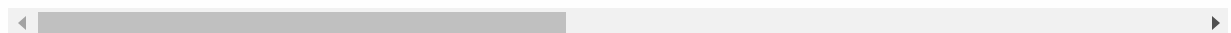
- JavaScript proporciona el método *bind* para asociar el objeto que será referenciado internamente.

```
/*
```

```
Dentro del método mostrarMensaje,  
(instancia de la clase Boton). con  
caso el texto 'Botón pulsado'
```

```
*/
```

```
buttonElement.addEventListener('click
```



- Un posible ejemplo de decorador de un método consiste en realizar la unión con *bind* desde la función decoradora.

```
/*
```

```
Función para decorar el método de  
la función no son utilizados para  
se nombran con la partícula _)
```

```
*/
```

```
function Autobind(_: any, __: string
```

```
// Referencia al método que se dec
```

```
const metodoOriginal = descriptor.  
console.log(metodoOriginal);
```

```
// Nuevo descriptor para el método
```

```
const nuevoDescriptor: PropertyDes  
configurable: descriptor.configu
```

```
... ..
```

```
enumerable: descriptor.enumerabl
```

```
/*
```

```
Este método es invocado solame  
clase Mostrar
```

```
*/
```

```
get() {
```

```
    console.log('Obtención del nue
```

```
// El objeto this hace referen
```

```
    console.log(this);
```

```
/*
```

```
El método que se decora se u  
(internamente this apuntará
```

```
*/
```

```
.....
```

```
        return metodoOriginal.bind(this)
    }
}
```

```
// Se devuelve el nuevo descriptor
return nuevoDescriptor;
}
```

```
class Boton {

    private mensaje = 'Botón pulsado';

    constructor() {

        // Botón pulsado
        console.log(this.mensaje);
    }
}
```

```
}

// Aplicación del decorador
@Autobind
mostrarMensaje() {
    console.log(this.mensaje);
}
}
```

```
const boton = new Boton();
const buttonElement = document.getEl
```

```
/*
```

```
Ahora no es necesario invocar al m
boton (instancia de la clase Boton
*/
```

```
buttonElement.addEventListener('clic
```



Módulos

- Modularización en TypeScript
- Introducción a los módulos
- Módulos en el frontend
- Módulos en el backend
- Funcionamiento de la exportación/importación

Modularización en TypeScript

- En TypeScript existen dos formas de modularizar o separar el código en archivos distintos.

- Mediante espacios de nombre: utiliza la sintaxis de *namespaces* para agrupar código, compilando por archivo o por empaquetado mediante la propiedad *outFile* en el archivo *tsconfig.json*.
- Mediante módulos: utiliza la sintaxis de importaciones/exportaciones de ECMAScript 6. La compilación se produce por archivo, pero las importaciones se realizan por separado. Existe la posibilidad de empaquetar múltiples archivos con empaquetadores como *webpack*.
- El uso de espacios de nombre está desaconsejado por *TSLint* y está recomendando el uso de módulos ECMAScript como forma estándar de modularizar el código. El famoso libro *TypeScript Deep Dive* también *desaconseja su uso*, así como

diferentes páginas de [StackOverflow](#) y la propia documentación de [TypeScript](#).

Introducción a los módulos

- La forma recomendada de utilizar módulos es mediante la sintaxis de ECMAScript 6, es decir, con *import* y *export*.
- La configuración necesaria y el código de TypeScript para soportar módulos es ligeramente distinto en aplicaciones del frontend y del backend. Los archivos mínimos a tener en cuenta son los siguientes:
 - Archivo *package.json*.
 - Archivo de configuración *tsconfig.json*.

- Archivo de código TypeScript donde se realiza la exportación.
- Archivo de código TypeScript donde se realiza la importación.
- Archivo HTML donde se cargan los archivos de JavaScript (solamente para aplicaciones del frontend).

Módulos en el frontend

- Archivo *package.json*: se debe incluir la propiedad *type* con el valor *module*.

```
/* package.json */  
{
```

```
/* ... */
```

```
"type": "module",  
  
/* ... */  
}
```

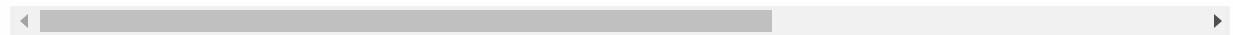
- Archivo de configuración *tsconfig.json*: se debe incluir la propiedad *target* con el valor *es6* (ECMAScript 6) o superior.

```
/* tsconfig.json */  
{  
  /* ... */  
  
  "target": "es6",  
  
  /* ... */  
}
```

- Archivo de código TypeScript donde se realiza la exportación: se debe incluir la partícula *export* en todas las declaraciones (función, variable, tipo, objeto, clase o interfaz) que se pretenden exportar.

// operaciones.ts

```
export function sumar(numero1: number, numero2: number): number {  
    return numero1 + numero2;  
}
```



- Archivo de código TypeScript donde se realiza la importación: se debe incluir la partícula *import*, el elemento que se pretende importar, la partícula *from* y, por último, la ruta al archivo donde se ha realizado la exportación (incluyendo la extensión *js* de los archivos de JavaScript).

// main.ts

```
import { sumar } from './operaciones
```

```
const resultado = sumar(1, 2);
```

```
console.log(resultado);
```



- Archivo HTML donde se cargan los archivos de JavaScript: se debe agregar el atributo *type* con el valor *module* a los dos archivos de JavaScript anteriores una vez ya compilados.

```
<!-- index.html -->
```

```
<script type="module" src="operacion
```

```
<script type="module" src="main.js">
```



- El programa debe ejecutarse bajo un servidor web (en Visual Studio Code puede utilizarse la

extensión [Live Server](#) para ello), dado que los módulos en JavaScript en una aplicación del frontend no pueden ejecutarse abriendo directamente el archivo HTML en un navegador.

- Sin embargo, este enfoque no es el más apropiado en producción dado que cada archivo de JavaScript es descargado por el cliente de forma independiente (ver pestaña *Network* de la consola de Chrome), aunque generalmente se realiza de forma paralela. Por tanto, a mayor número de archivos de JavaScript, mayor tiempo de carga de la página.
- Otro problema de los módulos en aplicaciones del frontend es la necesidad de añadir la extensión *js* en las importaciones.

- Para evitar este tipo de problemas con los módulos, es preferible hacer uso de algún framework del frontend ([React](#), [Angular](#), [Vue](#) o [Svelte](#), entre lo más importantes) o, al menos, empaquetadores como [Webpack](#).

Módulos en el backend

- Archivo *package.json*: no se debe incluir la propiedad *type* asignada al valor *module* como en las aplicaciones del frontend.
- Archivo de configuración *tsconfig.json*: se debe incluir la propiedad *target* con el valor *es6* (ECMAScript 6) o superior, la propiedad *module* al valor *commonjs* y la propiedad *esModuleInterop* al valor *true*.

```
/* tsconfig.json */
{
  /* ... */

  "target": "es6",
  "module": "commonjs",
  "esModuleInterop": true,

  /* ... */
}
```

- Archivo de código TypeScript donde se realiza la exportación: igual que en el frontend.

```
// operaciones.ts
```

```
export function sumar(numero1: number, numero2: number): number {
  return numero1 + numero2;
}
```

»



- Archivo de código TypeScript donde se realiza la importación: igual que en el backend, pero no debe incluirse la extensión *js* de los archivos JavaScript.

```
// main.ts
```

```
import { sumar } from './operaciones
```

```
const resultado = sumar(1, 2);
```

```
console.log(resultado);
```



Funcionamiento de la exportación/importación

- Existen varias formas de importar/exportar módulos en TypeScript en función de las necesidades del programador.
- Con respecto a la exportación:
 - Se utiliza la palabra reservada *export* delante de la declaración (función, variable, tipo, objeto, clase o interfaz) que pretenda exportarse.
 - Puede utilizarse la palabra clave reservada *default* (solamente una vez por archivo) para establecer una declaración como exportación por defecto.
 - Puede realizarse la exportación al final del archivo utilizando *export*, seguido del nombre de la declaración o declaraciones entre llaves.

- Algunos programadores optan por realizar todas las exportaciones en un archivo especial de nombre *index.ts*, que facilita la importación.
- Ejemplo de exportación de dos funciones en un archivo de TypeScript, utilizando una exportación al final del archivo y una exportación por defecto:

```
// hola.ts
```

```
function holaEN(nombre: string){  
  console.log(`Hello ${nombre}!`);  
}
```

```
function holaES(nombre: string){  
  console.log(`Hola ${nombre}!`);  
}
```

// Exportación al final del archivo

```
export { holaES };
```

*/**

Exportación por defecto de la func

una vez por archivo)

**/*

```
export default holaEN;
```




- Ejemplo de exportación, en un archivo *index.ts*, utilizando la palabra reservada *export* delante de la declaración:

// saludo/index.ts

// Exportación de la función saludo

```
export function saludo(nombre: string) {
```

```
console.log(`Saludos ${nombre}!`);  
}
```



- Con respecto a la importación:
 - Se realiza con la palabra reservada *import*, seguido del nombre de las declaraciones (entre llaves) que pretendan importarse y, a continuación, *from* y seguidamente el nombre del archivo donde se encuentran las declaraciones.
 - No se deben utilizar las llaves si se pretende importar la declaración por defecto (*default* en el archivo donde se exporta).
 - Pueden importarse todas las exportaciones de un archivo y almacenarlas en un objeto mediante la sintaxis ** as*, seguido del nombre

del objeto que agrupará todas las declaraciones exportadas.

- Puede utilizarse un alias de cualquier declaración mediante la partícula *as*.

- Ejemplo de importaciones específicas:

```
// main.ts
```


```
// Importa las dos funciones de hola
```

```
import { holaEN, holaES } from './ho
```

```
// Hello World!
```

```
holaEN('World');
```

```
// Hola Mundo!  
holaES( 'Mundo' );
```



- Ejemplo de importación por defecto:

```
// main.ts  
  
/*  
    Importa la exportación por defecto  
    (nombre arbitrario)  
*/  
import defaultHello from './hola';  
  
// Hello World!  
defaultHello( 'World' );
```




- Ejemplo de importación de todas las exportaciones de un archivo:

```
// main.ts
```

```
// Importa todas las exportaciones d  
import * as objeto from './hola';
```

```
// Hello World!  
objeto.holaEN('World');
```

```
// Hola Mundo!  
objeto.holaES('Mundo');
```



- Ejemplo de importación de un archivo index.ts.

```
// main.ts
```

```
/*
```

```
    Importa la función saludo del arch  
    indicar explícitamente ./saludo/in
```

```
    - - - - -
```

es especial y automáticamente Type

saludo)

**/*

```
import { saludo as s } from './salud
```

// Saludos Humano!

```
s( 'Humano' );
```



Otros

- Manipulación del DOM con TypeScript
- Librerías sin soporte para TypeScript
- TypeScript con *Node* y *Express*
- Otras herramientas

Manipulación del DOM con TypeScript

- Los elementos del **DOM** (**Document Object Model**) están tipados en TypeScript para una mejor manipulación y acceso a sus propiedades.

- Puede consultarse la [documentación de Mozilla](#) para obtener más información sobre los diferentes tipos disponibles para la manipulación del DOM.
- El método más importante para la obtención de elementos HTML desde JavaScript es *getElementById* (dentro del objeto *documento*), que en TypeScript es tipado como una función que retorna un valor de tipo *HTMLElement* | *null*.

```
const elementoHTML = document.getEle
```



- Para evitar problemas posteriores con valores nulos, es habitual utilizar el operador de aserción nulo (!)

```
const elementoHTML = document.getEle
```



- También es buena práctica realizar un casting considerando el tipo de elemento HTML que se obtiene, dado que *getElementById* siempre devuelve el valor *HTMLElement | null*

```
<!--
```

```
    Código HTML
```

```
-->
```

```
<form id="formulario">
```

```
    <!-- ... -->
```

```
</form>
```

```
/*
```

```
    Código de TypeScript
```

```
*/
```

```
const elementoHTML = document.getEle
```



- Los objetos asociados a elementos HTML están tipados en TypeScript como `HTML<nombre_etiqueta>Element`. En el ejemplo anterior, el elemento *form* de HTML es obtenido mediante *getElementById* y tipado como *HTMLFormElement*.

Librerías sin soporte para TypeScript

- A veces es necesario recurrir a librerías de JavaScript que no tienen soporte para TypeScript.

```
<!-- lib.js -->
```

<!-- Librería de JavaScript que apor

<script>

const **PI** = 3.14;

</script>



<!-- main.js -->

<!-- Código empaquetado con Webpack

<script type="module" **src**="lib.js"><



- Las variables declaradas en la librería existen en el contexto global, pero no son detectadas por TypeScript.

// main.ts

```
// Error porque la variable PI no es
// console.log(PI);
```

- Para estos casos, TypeScript proporciona la expresión *declare var* para declarar variables que existen en el contexto global pero que no pueden ser detectadas directamente.

```
// main.ts
```

```
/*
```

```
    Se indica a TypeScript que confíe
    tipo number
```

```
*/
```

```
declare var PI: number;
```

```
// 3.14
```

```
console.log(PI);
```

TypeScript con *Node* y *Express*

- Para comenzar un proyecto de TypeScript con *Node* y *Express*, en primer lugar puede comenzarse creando un nuevo proyecto *Node* con *npm*.

```
npm init
```

- A continuación, se instalan los paquetes *@types* para *Node* y *Express*.

```
# Instalación de Typescript forma gl
```

```
npm install typescript -g
```

```
# Agrega el tipado a todos los módulos
```

```
npm install @types/node --save-dev
```

```
# Instalación de express de su paquete
```

```
npm install @types/express express -
```

- Seguidamente es importante realizar las siguientes modificaciones en el archivo *tsconfig.json*:
 - *target: es6* (o superior)
 - *module: commonjs*
 - *moduleResolution: node*
 - *esModuleInterop: true*
 - Configurar directorios *outDir* y *rootDir*

```
/* tsconfig.json */
```

```
{  
  "compilerOptions": {  
    "target": "es2018",  
    "outDir": "dist",  
    "module": "commonjs",
```



```
    "esModuleInterop": true,  
    "moduleResolution": "node",  
    "outDir": "dist",  
    "rootDir": "src",  
  }  
}
```

- Para detectar el tipado de Express es importante que se importe con la palabra clave *import* y no con *require*.

// Importación del módulo Express

```
import express from 'express';
```

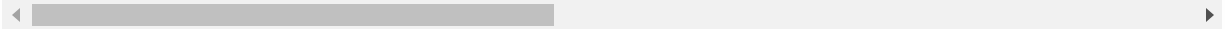
```
const app = express();
```

// Método GET

```
app.get('/', (req, res) => {  
    res.send('¡Hola mundo!')
```

```
});
```

```
app.listen(5000, () => console.log('
```



- Finalmente, puede utilizarse *nodemon* para facilitar la ejecución del programa, integrándolo dentro de *scripts* en el archivo *package.json*.

Compilación del archivo

```
tsc main.ts
```

Instalación de nodemon

```
npm install nodemon -g
```

Ejecución de la aplicación

```
nodemon main.js
```

- También existe el paquete [ts-node](#) para ejecutar archivos TypeScript con Node (concretamente con el comando *ts-node*). Solamente es recomendado para desarrollo y no para producción. Para producción es preferible hacer uso de [pm2](#) o paquetes similares.

Otras herramientas

- [Deno](#): es un entorno de ejecución para JavaScript y TypeScript (Node solamente lo es para JavaScript) basado en el motor v8 de Google.
- [tsc-multi](#): permite compilar varios proyectos de TypeScript al mismo tiempo.