

# **TYPESCRIPT**

**Para profesionales**



**Más de 80 páginas  
de consejos y trucos**



# Contenido

<b>Acerca de</b>	1
<b>Capítulo 1: Introducción a TypeScript</b>	2
Sección 1.1: Instalación y configuración	2
Sección 1.2: Sintaxis básica	4
Sección 1.3: Hola Mundo	5
Sección 1.4: Ejecución de TypeScript utilizando ts-node	6
Sección 1.5: TypeScript REPL en Node.js	6
<b>Capítulo 2: Por qué y cuándo usar TypeScript</b>	8
Sección 2.1: Seguridad	8
Sección 2.2: Legibilidad	8
Sección 2.3: Utillaje	8
<b>Capítulo 3: Tipos del núcleo de TypeScript</b>	9
Sección 3.1: Tipos literales de cadena	9
Sección 3.2: Tuple	12
Sección 3.3: Booleanos	12
Sección 3.4: Tipos de intersección	13
Sección 3.5: Tipos en argumentos de función y valor de retorno. Número	13
Sección 3.6: Tipos en argumentos de función y valor de retorno. Cadena	14
Sección 3.7: const Enum	14
Sección 3.8: Número	15
Sección 3.9: Cadena	15
Sección 3.10: Matriz	16
Sección 3.11: Enum	16
Sección 3.12: Cualquiera	16
Sección 3.13: Nulidad	16
<b>Capítulo 4: Matrices</b>	17
Sección 4.1: Encontrar un objeto en una matriz	17
<b>Capítulo 5: Enums</b>	18
Sección 5.1: Enums con valores explícitos	18
Sección 5.2: Cómo obtener todos los valores enum	19
Sección 5.3: Extensión de enums sin implementación de enum personalizada	19
Sección 5.4: Implementación de enum personalizados: extends para enums	19
<b>Capítulo 6: Funciones</b>	21
Sección 6.1: Parámetros opcionales y por defecto	21
Sección 6.2: Función como parámetro	21
Sección 6.3: Funciones con tipos de unión	23
Sección 6.4: Tipos de funciones	23
<b>Capítulo 7: Clases</b>	24
Sección 7.1: Clases abstractas	24
Sección 7.2: Clase simple	24
Sección 7.3: Herencia básica	25
Sección 7.4: Constructores	25
Sección 7.5: Accesos	26
Sección 7.6: Transpilación	27
Sección 7.7: Monkey patch una función en una clase existente	28
<b>Capítulo 8: Decorador de clases</b>	29
Apartado 8.1: Generación de metadatos mediante un decorador de clases	29
Sección 8.2: Pasar argumentos a un decorador de clase	29
Sección 8.3: Decorador básico de clases	30
<b>Capítulo 9: Interfaces</b>	32
Sección 9.1: Interfaz extensible	32
Sección 9.2: Interfaz de clase	32

Sección 9.3: Uso de interfaces para polimorfismo .....	33
Sección 9.4: Interfaces genéricas.....	34
Apartado 9.5: Añadir funciones o propiedades a una interfaz existente.....	35
Sección 9.6: Implementación implícita y forma del objeto.....	35
Sección 9.7: Uso de interfaces para reforzar tipos.....	36
<b>Capítulo 10: Genéricos .....</b>	<b>37</b>
Sección 10.1: Interfaces genéricas.....	37
Sección 10.2: Clase genérica.....	37
Sección 10.3: Parámetros de tipo como restricciones .....	38
Sección 10.4: Restricciones genéricas.....	38
Sección 10.5: Funciones genéricas .....	39
Sección 10.6: Uso de clases y funciones genéricas: .....	39
<b>Capítulo 11: Comprobación estricta de nulos.....</b>	<b>40</b>
Sección 11.1: Comprobación estricta de nulos en acción .....	40
Sección 11.2: Afirmaciones no nulas .....	40
<b>Capítulo 12: Guardas de tipo definidas por el usuario .....</b>	<b>42</b>
Sección 12.1: Funciones de protección de tipos.....	42
Apartado 12.2: Uso de instanceof .....	43
Sección 12.3: Uso de typeof .....	43
<b>Capítulo 13: Ejemplos básicos de TypeScript .....</b>	<b>45</b>
Apartado 13.1: 1 ejemplo básico de herencia de clases utilizando las palabras clave extends y super .....	45
Sección 13.2: 2 ejemplo de variable de clase estática - cuenta cuántas veces se invoca un método .....	45
<b>Capítulo 14: Importación de bibliotecas externas.....</b>	<b>46</b>
Sección 14.1: Búsqueda de archivos de definición .....	46
Sección 14.2: Importar un módulo desde npm.....	47
Sección 14.3: Utilización de bibliotecas externas globales sin tipados.....	47
Sección 14.4: Cómo encontrar archivos de definición con TypeScript 2.x .....	47
<b>Capítulo 15: Módulos: exportación e importación.....</b>	<b>49</b>
Sección 15.1: Módulo Hello world.....	49
Sección 15.2: Reexportación.....	49
Apartado 15.3: Declaraciones de exportación/importación .....	51
<b>Capítulo 16: Publicar archivos de definición de TypeScript .....</b>	<b>52</b>
Sección 16.1: Incluir archivo de definición con biblioteca en npm.....	52
<b>Capítulo 17: Uso de TypeScript con webpack .....</b>	<b>53</b>
Sección 17.1: webpack.config.js.....	53
<b>Capítulo 18: Mixins .....</b>	<b>54</b>
Sección 18.1: Ejemplo de Mixins .....	54
<b>Capítulo 19: Cómo utilizar una biblioteca JavaScript sin un archivo de definición de tipos .....</b>	<b>55</b>
Sección 19.1: Crear un módulo que exporte un valor por defecto any.....	55
Sección 19.2: Declarar un any global.....	55
Sección 19.3: Utilizar un módulo ambiente.....	56
<b>Capítulo 20: TypeScript instalación de typescript y ejecución del compilador de typescript tsc.....</b>	<b>57</b>
Sección 20.1: Pasos.....	57
<b>Capítulo 21: Configurar el proyecto typescript para compilar todos los archivos en typescript.....</b>	<b>59</b>
Sección 21.1: Configuración del archivo de configuración de TypeScript.....	59
<b>Capítulo 22: Integración con herramientas de compilación.....</b>	<b>61</b>
Sección 22.1: Browserify.....	61
Sección 22.2: Webpack.....	61
Sección 22.3: Grunt.....	62
Sección 22.4: Gulp.....	62
Sección 22.5: MSBuild .....	63
Sección 22.6: NuGet.....	63

Sección 22.7: Instalar y configurar los cargadores webpack + .....	64
<b>Capítulo 23: Uso de TypeScript con RequireJS</b> .....	65
Sección 23.1: Ejemplo HTML usando RequireJS CDN para incluir un archivo TypeScript ya compilado .....	65
Sección 23.2: tsconfig.json ejemplo para compilar en la carpeta view usando el estilo de importación RequireJS .....	65
<b>Capítulo 24: TypeScript con AngularJS</b> .....	66
Sección 24.1: Directiva .....	66
Apartado 24.2: Ejemplo sencillo .....	67
Sección 24.3: Componente .....	67
<b>Capítulo 25: TypeScript con SystemJS</b> .....	69
Sección 25.1: Hello World en el navegador con SystemJS .....	69
<b>Capítulo 26: Uso de TypeScript con React (JS y nativo)</b> .....	72
Sección 26.1: Componente ReactJS escrito en TypeScript .....	72
Sección 26.2: TypeScript & react & webpack .....	73
<b>Capítulo 27: TSLint - asegurando la calidad y consistencia del código</b> .....	75
Sección 27.1: Configuración para reducir los errores de programación .....	75
Sección 27.2: Instalación y configuración .....	75
Sección 27.3: Conjuntos de reglas TSLint .....	76
Sección 27.4: Configuración básica de tslint.json .....	76
Apartado 27.5: Utilizar un conjunto de reglas predefinido como predeterminado .....	76
<b>Capítulo 28: tsconfig.json</b> .....	78
Sección 28.1: Crear proyecto TypeScript con tsconfig.json .....	78
Sección 28.2: Configuración para reducir los errores de programación .....	79
Sección 28.3: compileOnSave .....	80
Sección 28.4: Observaciones .....	80
Sección 28.5: preserveConstEnums .....	81
<b>Capítulo 29: Depuración</b> .....	82
Sección 29.1: TypeScript con ts-node en WebStorm .....	82
Sección 29.2: TypeScript con ts-node en Visual Studio Code .....	83
Sección 29.3: JavaScript con SourceMaps en Visual Studio Code .....	84
Sección 29.4: JavaScript con SourceMaps en WebStorm .....	84
<b>Capítulo 30. Pruebas unitarias Pruebas unitarias</b> .....	86
Sección 30.1: Cinta .....	86
Sección 30.2: jest (ts-jest) .....	87
Sección 30.3: Alsaciano .....	89
Sección 30.4: complemento chai-immutable .....	89
<b>Créditos</b> .....	91
<b>También le puede gustar</b> .....	93

Por favor, siéntase libre de compartir este PDF con cualquier persona de forma gratuita, la última versión de este libro se puede descargar desde:

<https://goalkicker.com/TypeScriptBook>

Este libro de *Notas de TypeScript para Profesionales* está compilado a partir de la [Documentación de](#) Stack Overflow, el contenido está escrito por la hermosa gente de Stack Overflow. El contenido del texto está liberado bajo Creative Commons BY-SA, ver los créditos al final de este libro de quién contribuyó a los distintos capítulos. Las imágenes pueden ser copyright de sus respectivos propietarios a menos que se especifique lo contrario.

Este es un libro gratuito no oficial creado con fines educativos y no está afiliado con grupo(s) o compañía(s) oficial(es) de TypeScript ni Stack Overflow. Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos propietarios.

No se garantiza que la información presentada en este libro sea correcta ni exacta. Utilícelo bajo su propia responsabilidad.

Envíe sus comentarios y correcciones [web@petercv.com](mailto:web@petercv.com)

# Capítulo 1: Introducción a TypeScript

Versión	Fecha de publicación
<a href="#">2.8.3</a>	2018-04-20
<a href="#">2.8</a>	2018-03-28
<a href="#">2.8 RC</a>	2018-03-16
<a href="#">2.7.2</a>	2018-02-16
<a href="#">2.7.1</a>	2018-02-01
<a href="#">2.7 beta</a>	2018-01-18
<a href="#">2.6.1</a>	2017-11-01
<a href="#">2.5.2</a>	2017-09-01
<a href="#">2.4.1</a>	2017-06-28
<a href="#">2.3.2</a>	2017-04-28
<a href="#">2.3.1</a>	2017-04-25
<a href="#">2.3.0 beta</a>	2017-04-04
<a href="#">2.2.2</a>	2017-03-13
<a href="#">2.2</a>	2017-02-17
<a href="#">2.1.6</a>	2017-02-07
<a href="#">2.2 beta</a>	2017-02-02
<a href="#">2.1.5</a>	2017-01-05
<a href="#">2.1.4</a>	2016-12-05
<a href="#">2.0.8</a>	2016-11-08
<a href="#">2.0.7</a>	2016-11-03
<a href="#">2.0.6</a>	2016-10-23
<a href="#">2.0.5</a>	2016-09-22
<a href="#">2.0 Beta</a>	2016-07-08
<a href="#">1.8.10</a>	2016-04-09
<a href="#">1.8.9</a>	2016-03-16
<a href="#">1.8.5</a>	2016-03-02
<a href="#">1.8.2</a>	2016-02-17
<a href="#">1.7.5</a>	2015-12-14
<a href="#">1.7</a>	2015-11-20
<a href="#">1.6</a>	2015-09-11
<a href="#">1.5.4</a>	2015-07-15
<a href="#">1.5</a>	2015-07-15
<a href="#">1.4</a>	2015-01-13
<a href="#">1.3</a>	2014-10-28
<a href="#">1.1.0.1</a>	2014-09-23

## Sección 1.1: Instalación y configuración

### Fondo

TypeScript es un superconjunto tipado de JavaScript que se compila directamente con código JavaScript. Los archivos TypeScript suelen utilizar la extensión `.ts`. Muchos IDE admiten TypeScript sin necesidad de ninguna otra configuración, pero TypeScript también se puede compilar con el paquete TypeScript Node.JS desde la línea de comandos.

## IDEs

### Visual Studio

- Visual Studio **2015** incluye TypeScript.
- Visual Studio **2013 Update 2** o posterior incluye TypeScript, o puedes [descargar TypeScript para versiones anteriores](#).

### Código de Visual Studio

- [Visual Studio Code](#) (vscode) proporciona autocompletado contextual, así como herramientas de refactorización y depuración para TypeScript. vscode está implementado en TypeScript. Disponible para Mac OS X, Windows y Linux.

### WebStorm

- [WebStorm 2016.2](#) viene con TypeScript y un compilador integrado. [WebStorm no es gratuito].

### IntelliJ IDEA

- [IntelliJ IDEA 2016.2](#) tiene soporte para TypeScript y un compilador a través de un [plugin](#) mantenido por el equipo de JetBrains. [IntelliJ no es gratuito].

### Atom y atom-typescript

- [Atom](#) soporta TypeScript con el paquete [atom-typescript](#).

### Texto Sublime

- [Sublime](#) Text soporta TypeScript con el paquete [TypeScript](#).

Instalar la interfaz de línea de comandos

Instalar [Node.js](#)

Instalar el paquete npm globalmente

Puedes instalar TypeScript globalmente para tener acceso a él desde cualquier directorio.

```
npm install -g typescript
```

o

Instale el paquete npm localmente

Puedes instalar TypeScript localmente y guardarlo en package.json para restringirlo a un directorio.

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

 Canales de instalación

Se puede instalar desde:

- Canal estable: `npm install typescript`
- Canal beta: `npm install typescript@beta`
- Canal de desarrollo: `npm install typescript@next`

### Compilación de código TypeScript

El comando de compilación `tsc` viene con `typescript`, que puede utilizarse para compilar código.

```
tsc mi-codigo.ts
```

Esto crea un archivo `my-code.js`.

Compilar con `tsconfig.json`



También puede proporcionar opciones de compilación que viajan con su código a través de un archivo `tsconfig.json`. Para iniciar un nuevo proyecto TypeScript, entra en el directorio raíz de tu proyecto en una ventana de terminal y ejecuta `tsc --init`. Este comando generará un archivo `tsconfig.json` con opciones de configuración mínimas, similar al de abajo.

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

Con un archivo `tsconfig.json` colocado en la raíz de su proyecto TypeScript, puede utilizar el comando `tsc` para ejecutar la compilación.

## Sección 1.2: Sintaxis básica

TypeScript es un superconjunto tipado de JavaScript, lo que significa que todo el código JavaScript es código TypeScript válido. Además, TypeScript añade un montón de nuevas características.

TypeScript hace que JavaScript se parezca más a un lenguaje fuertemente tipado y orientado a objetos, similar a C# y Java. Esto significa que el código TypeScript tiende a ser más fácil de usar para grandes proyectos y que el código tiende a ser más fácil de entender y mantener. La fuerte tipificación también significa que el lenguaje puede (y es) precompilado y que a las variables no se les pueden asignar valores que estén fuera de su rango declarado. Por ejemplo, cuando una variable TypeScript se declara como un número, no se puede asignar un valor de texto.

Esta fuerte tipificación y orientación a objetos hace que TypeScript sea más fácil de depurar y mantener, y esos eran dos de los puntos más débiles del JavaScript estándar.

Declaraciones de tipo

Puede añadir declaraciones de tipo a variables, parámetros de funciones y tipos de retorno de funciones. El tipo se escribe después de dos puntos tras el nombre de la variable, así: `var num: number = 5;` El compilador comprobará los tipos (cuando sea posible) durante la compilación e informará de los errores de tipo.

```
var num: número = 5;
num = "esto es una cadena"; // error: El tipo 'cadena' no es assignable al tipo 'número'.
```

Los tipos básicos son :

- número (tanto enteros como de coma flotante)
- cadena
- `boolean`
- Matriz. Puede especificar los tipos de los elementos de un array. Existen dos formas equivalentes de definir los tipos de matrices:
  - Matriz `T[]` y `T[]`. Por ejemplo:
    - `number[]` - matriz de números
    - `Matriz< cadena >` - matriz de cadenas
- Tuplas. Las tuplas tienen un número fijo de elementos con tipos específicos.
  - `[boolean, string]` - tupla en la que el primer elemento es un boolean y el segundo es una cadena.
  - `[número, número, número]` - tupla de tres números.

- `{}` - objeto, puede definir sus propiedades o indexador
  - `{ nombre: cadena, edad: número }` - objeto con atributos de nombre y edad
  - `{ [clave: cadena]: número }` - un diccionario de números indexados por
- cadena `enum` - `{ Rojo= 0, Azul, Verde }` - enumeración asignada a números
- Función. Se especifican los tipos de los parámetros y del valor de retorno:
  - `(param: n número) => cadena` - función que toma un parámetro numérico y devuelve una cadena
  - `() => number` - función sin parámetros que devuelve un número.
  - `(a: cadena, b?: booleano) => void` - función que toma una cadena y opcionalmente un booleano sin valor de retorno.
- `any` - Permite cualquier tipo. Las expresiones que incluyen `any` no se comprueban.
- `void` - representa "nada", puede utilizarse como valor de retorno de una función. Sólo `null` e `indefinido` forman parte de tipo de `vacío`.
- `never`
  - `let foo: never;` - Como el tipo de variables bajo guardas de tipo que nunca son verdaderas.
  - `function error(mensaje: cadena): never { throw new Error(mensaje); }` - Como tipo de retorno de funciones que nunca devuelven.
- `null` - tipo para el valor `null`. `null` forma parte implícitamente de todos los tipos, a menos que se active la comprobación estricta de `null`.

## Fundición

Por ejemplo, puede realizar una selección explícita mediante corchetes angulares:

```
var derivado: MyInterface; (<ImplementingClass>
derived).someSpecificMethod();
```

Este ejemplo muestra una clase derivada que es tratada por el compilador como una `MyInterface`. Sin el casting en la segunda línea el compilador lanzaría una excepción ya que no entiende `someSpecificMethod()`, pero el casting a través de `<ImplementingClass>derived` sugiere al compilador qué hacer.

Otra forma de lanzar en TypeScript es utilizando la palabra clave `as`:

```
var derivado: MiInterfaz;
(derivado como ImplementingClass).someSpecificMethod();
```

Desde TypeScript 1.6, por defecto se usa la palabra clave `as`, porque usar `<>` es ambiguo en archivos `.jsx`. Esto se menciona en [la documentación oficial de TypeScript](#).

## Clases

Las clases se pueden definir y utilizar en el código TypeScript. Para obtener más información sobre las clases, consulta la página de documentación Clases.

## Sección 1.3: Hola Mundo

```
class Greeter {
  saludo: cadena;

  constructor(mensaje: cadena) {
    this.greeting = mensaje;
  }
  greet(): cadena {
    devolver este.saludo;
  }
};
```

```
let greeter= new Greeter(" ¡Hola, mundo!");
console.log(greeter.greet());
```

Aquí tenemos una clase, Greeter , que tiene un constructor y un método greet . Podemos construir una instancia de la clase utilizando la palabra clave new y pasar una cadena que queremos que el método greet envíe a la . La instancia de nuestra clase Greeter se almacena en la variable greeter que luego usamos para llamar al método greet .

## Sección 1.4: Ejecutar TypeScript usando ts-node

[ts-node](#) es un paquete npm que permite al usuario ejecutar archivos typescript directamente, sin necesidad de precompilación usando tsc . También proporciona [REPL](#).

Instale ts-node globalmente utilizando

```
npm install -g ts-node
```

ts-node no incluye el compilador typescript, por lo que es posible que tenga que .

```
npm install -g typescript
```

Ejecución del script

Para ejecutar un script llamado *main.ts*, ejecute

```
ts-node main.ts
```

```
// main.ts
console.log("Hola mundo");
```

Ejemplo de uso

```
$ ts-node main.ts Hola
mundo
```

Ejecución de REPL

Para ejecutar REPL ejecute el comando ts-node

Ejemplo de uso

```
$ node-ts
> const suma= (a, b): número=> a+ b;
indefinido
> suma(2, 2)
4
> .salir
```

Para salir de REPL utilice el comando `.exit` o pulse CTRL+C dos veces.

## Sección 1.5: TypeScript REPL en Node.js

Para usar TypeScript REPL en Node.js puedes usar [el paquete tsun](#).

Instálalo globalmente con

```
npm install -g tsun
```

y ejecutar en su terminal o símbolo del sistema con `tsun` comando

Ejemplo de uso:

```
$ tsun
TSUN : Nodo actualizado TypeScript
tipo en TypeScript expresi ón a evaluar tipo
:help para comandos en repl
$ function multiplicar(x, y) {
.. devuelve x * y;
..}
indefinido
$ multiplicar(3, 4)
12
```

# Capítulo 2: Por qué y cuándo usar TypeScript

Si encuentras persuasivos los argumentos a favor de los sistemas de tipos en general, entonces estarás contento con TypeScript.

Aporta muchas de las ventajas de los sistemas de tipos (seguridad, legibilidad, herramientas mejoradas) al ecosistema JavaScript. También sufre algunos de los inconvenientes de los sistemas de tipos (complejidad añadida e incompletitud).

## Sección 2.1: Seguridad

TypeScript detecta los errores de tipo de forma temprana a través del análisis estático:

```
función double(x: n número): n número {  
  devuelve 2 * x;  
}  
double('2');  
//      ~~~ Argumento de tipo "'2'" no es assignable a parámetro de tipo 'number'.
```

## Sección 2.2: Legibilidad

TypeScript permite a los editores proporcionar documentación contextual:

```
'foo'.slice()  
slice(start?: number, end?: number): string  
The index to the beginning of the specified portion of stringObj.  
Returns a section of a string.
```

Nunca volverás a olvidar si String. `prototype.slice` toma (inicio, parada) o (inicio, longitud).

## Sección 2.3: Utilaje

TypeScript permite a los editores realizar refactorizaciones automatizadas que conocen las reglas de los lenguajes.

```
let foo = '123';  
  
{  
  const foo = (x: number) => {  
    return 2 * x;  
  }  
  
  foo(2);  
}
```

Aquí, por ejemplo, Visual Studio Code es capaz de renombrar las referencias al `foo` interno sin alterar el `foo` externo. Esto sería difícil de hacer con un simple buscar/reemplazar.

# Capítulo 3: Tipos del núcleo de TypeScript

## Sección 3.1: Tipos literales de cadena

Los tipos literales de cadena permiten especificar el valor exacto que puede tener una cadena.

```
let miMascotaFavorita:
"perro"; miMascotaFavorita=
"perro";
```

Cualquier otra cadena dará error.

```
// Error: El tipo "'roca'" no es assignable al tipo "'perro'".
// miMascotaFavorita= "roca";
```

Junto con los alias de tipo y los tipos de unión se obtiene un comportamiento similar al de los enum.

```
tipo Especie= "gato" | "perro" | "pájaro";

function comprarMascota(mascota: Especie, nombre: cadena) :
Pet { /*...*/ } buyPet(myFavoritePet /* "perro" como se define arriba */,
"Rocky");

// Error: Argumento de tipo "'roca'" no es assignable a parámetro de tipo "'gato' | 'perro' | 'pájaro'". El
tipo "'roca'" no es assignable al tipo "'pájaro'".
// buyPet("rock", "Rocky");
```

Los tipos literales de cadena pueden utilizarse para distinguir sobrecargas.

```
function comprarMascota(mascota: Especie, nombre:
cadena) : Mascota; function comprarMascota(mascota:
"gato", nombre: cadena): Gato; function
comprarMascota(mascota: "perro", nombre: cadena):
Perro; function buyPet(pet: "pájaro", nombre: cadena):
Pájaro;
function comprarMascota(mascota: Especie, nombre: cadena) : Mascota { /*...*/ }

let perro= buyPet(miMascotaFavorita /* "perro" como se define arriba */, "Rocky");
// perro es de tipo Perro (perro: Perro)
```

Funcionan bien para guardias de tipo definidas por el usuario.

```
interfaz Mascota {
    especie: Especie;
    comer();
    dormir();
}

interfaz Cat extends Pet {
    especie: "gato";
}

interfaz Bird extends Pet {
    especie: "pájaro";
    sing();
}

function petIsCat(pet: Mascota): pet es Gato {
    return pet.species=== "gato";
}
```

```
function petIsBird(pet: Pet): pet es Pájaro {
    return pet.species === "pájaro";
}

function playWithPet(pet: Pet){
    if(petIsCat(pet)) {
        // mascota es ahora de tipo Gato (mascota: Gato)
        pet.eat();
        pet.sleep();
    } else if(petIsBird(pet)) {
        // mascota es ahora de tipo Pájaro (mascota: Pájaro)
        pet.eat();
        pet.sing();
        pet.sleep();
    }
}
```

Código de ejemplo completo

```
let miMascotaFavorita: "perro";
miMascotaFavorita = "perro";

// Error: El tipo "'roca'" no es assignable al tipo "'perro'".
// miMascotaFavorita = "roca";

type Especie = "gato" | "perro" | "pájaro";

interface Mascota {
    especie: Especie;
    nombre: cadena;
    eat();
    walk();
    dormir();
}

interfaz Cat extends Pet { especie:
    "gato";
}

interfaz Dog extends Pet { especie:
    "perro";
}

interfaz Bird extends Pet { especie:
    "pájaro";
    sing();
}

// Error: La interfaz 'Roca' extiende incorrectamente la interfaz 'Mascota'. Los tipos de la propiedad 'species'
// son incompatibles. El tipo "'roca'" no es assignable al tipo "'gato' | 'perro' | 'pájaro'". El tipo "'roca'" no es
// assignable al tipo "'pájaro'".
// interfaz Rock extends Pet {
//     tipo: "roca";
// }

function comprarMascota(mascota: Especie, nombre: cadena)
: Mascota; function comprarMascota(mascota: "gato",
nombre: cadena): Gato; function comprarMascota(mascota:
"perro", nombre: cadena): Perro; function buyPet(pet:
"pájaro", nombre: cadena): Pájaro; function
buyPet(mascota: Especie, nombre: cadena) : Mascota {
    if(mascota === "gato") {
```

```

    devolver {
      especie: "gato",
      nombre: nombre,
      eat: function () { console.log(`$ {this.name}
        eats.`);
      }, walk: function () { console.log(`$
        {this.name} paseos.`);
      }, sleep: function () { console.log(`$
        {this.name} duerme.`);
      }
    } como Cat;
  } else if(mascota=== "perro") {
    devolver {
      especie: "perro",
      nombre: nombre,
      eat: function () { console.log(`$ {this.name}
        eats.`);
      }, walk: function () { console.log(`$
        {this.name} paseos.`);
      }, sleep: function () { console.log(`$
        {this.name} duerme.`);
      }
    } como Perro;
  } else if(mascota=== "pájaro") {
    devolver {
      especie: "pájaro",
      nombre: nombre,
      eat: function () { console.log(`$ {this.name}
        eats.`);
      }, walk: function () { console.log(`$
        {this.name} paseos.`);
      }, sleep: function () { console.log(`$
        {this.name} duerme.`);
      }, sing: function () { console.log(`$
        {this.name} canta.`);
      }
    } como Bird;
  } else {
    throw `Lo sentimos, no disponemos de$ {pet}. ¿Le gustaría comprar un perro?`;
  }
}

function petIsCat(pet: Mascota): pet es Gato {
  return pet.species=== "gato";
}

function petIsDog(pet: Mascota): pet es Perro {
  return mascota.especie=== "perro";
}

function petIsBird(pet: Pet): pet es Pájaro {
  return pet.species=== "pájaro";
}

function playWithPet(pet: Pet) { console.log(`Hey$
  {pet.name}, vamos a jugar.`);

  if(petIsCat(pet)) {
    // mascota es ahora de tipo Gato (mascota: Gato)

    pet.eat();
    pet.sleep();
  }
}

```



```

    // Error: El tipo "'pájaro'" no es assignable al tipo "'gato'".
    // pet.type= "pájaro";

    // Error: La propiedad 'sing' no existe en el tipo 'Cat'.
    // pet.sing();

} else if(petIsDog(pet)) {
    // mascota es ahora de tipo Perro (mascota: Perro)

    pet.eat();
    pet.walk();
    pet.sleep();

} else if(petIsBird(pet)) {
    // mascota es ahora de tipo Pájaro (mascota: Pájaro)

    pet.eat();
    pet.sing();
    pet.sleep();
} else {
    throw "Una desconocida. ¿Compraste una piedra?";
}
}

let perro= buyPet(miMascotaFavorita /* "perro" como se define arriba */, "Rocky");
// perro es de tipo Perro (perro: Perro)

// Error: Argumento de tipo "'roca'" no es assignable a parámetro de tipo "'gato'| 'perro'| 'pájaro'". El
// tipo "'roca'" no es assignable al tipo "'pájaro'".
// buyPet("rock", "Rocky");

playWithPet(perro);
// Salida: Hey Rocky, vamos a jugar.
//         Rocky come.
//         Rocky camina.
//         Rocky duerme.

```

## Sección 3.2: Tupla

Tipo de matriz con tipos conocidos y posiblemente diferentes:

```

dejar día: [n número, cadena];
día= [0, 'Lunes'];           // válido
día= ['cero', 'lunes'];      // inválido: 'cero' no es numérico
console.log(día[0]);         // 0
console.log(día[1]);         // Lunes

day[2]= 'Sábado';           // válido: [0, 'Sábado']
día[3]= false;              // invalid: must be union type of 'number| string'

```

## Sección 3.3: Booleanos

Un booleano representa el tipo de dato más básico en TypeScript, con el propósito de asignar valores verdadero/falso.

```

// establecer con valor inicial (verdadero o falso)
let isTrue: boolean= true;

// por defecto 'undefined', cuando no se establece explícitamente
let unsetBool: boolean;

```

// también se puede establecer en "null"

```
let nullableBool: boolean = null;
```

## Sección 3.4: Tipos de intersección

Un Tipo de intersección combina el miembro de dos o más tipos.

```
interfaz Cuchillo {
    cut();
}

interfaz BottleOpener{
    abrirBotella();
}

interfaz Destornillador{
    turnScrew();
}

tipo SwissArmyKnife= Cuchillo & Abrebotellas & Destornillador;

function use(tool: SwissArmyKnife){
    console.log(" ¡Puedo hacer cualquier cosa!");

    tool.cut();
    tool.openBottle();
    tool.turnScrew();
}
```

## Sección 3.5: Tipos en argumentos de función y valor de retorno. Número

Al crear una función en TypeScript, puede especificar el tipo de datos de los argumentos de la función y el tipo de datos del valor de retorno.

Ejemplo:

```
function suma(x: n número, y: n número): n número {
    devolver x+ y;
}
```

Aquí la sintaxis `x: n número, y: n número` significa que la función puede aceptar dos argumentos `x` e `y` y sólo pueden ser números y `(...): n número {` significa que el valor de retorno sólo puede ser un número

Uso:

```
sum(84+ 76) // será devolver 160
```

Nota:

No puede hacerlo

```
function suma(x: cadena, y: cadena): n número {
    devolver x+ y;
}
```

o

```
function suma(x: número, y: n número): cadena {
    devolver x+ y;
}
```

recibirá los siguientes errores:

error TS2322: El tipo 'string' no es asignable al tipo 'number' y error TS2322: Type 'number' is not assignable to type 'string' respectively

## Sección 3.6: Tipos en argumentos de función y valor de retorno. Cadena

Por ejemplo:

```
function hola(nombre: cadena): cadena {
    return `¡Hola$ {nombre}!`;
}
```

Aquí la sintaxis nombre: cadena significa que la función puede aceptar un argumento de nombre y este argumento sólo puede ser una cadena y (...): cadena { significa que el valor de retorno sólo puede ser una cadena

Uso:

```
hello(' DocumentaciónStackOverflow') // será return ¡Hola DocumentaciónStackOverflow!
```

## Sección 3.7: const Enum

Un Enum const es lo mismo que un Enum normal. Excepto que no se genera ningún objeto en tiempo de compilación. En su lugar, los valores literales se sustituyen donde se utiliza la const Enum.

*// TypeScript: Un const Enum puede definirse como un Enum normal (con valor inicial, valores específicos, etc.)*

```
const enum NinjaActivity {
    Espionaje,
    Sabotaje,
    asesinato
}
```

*// JavaScript: Pero no se genera nada*

*// TypeScript: Excepto si lo usas*

```
let miActividadNinjaFavorita= NinjaActivity.Espionaje;
console.log(miActividadPirataFavorita); // 0
```

*// JavaScript: A continuación, sólo el número del valor se compila en el código*

```
// var myFavoriteNinjaActivity= 0 /* Espionaje */;
// console.log(miActividadPirataFavorita); // 0
```

*// TypeScript: Lo mismo para el otro ejemplo constante*

```
console.log(NinjaActivity["Sabotaje"]); // 1
```

*// JavaScript: Sólo el número y en un comentario el nombre del valor*

```
// console.log(1 /* "Sabotaje" */); // 1
```

*// TypeScript: Pero sin el objeto no es posible el acceso en tiempo de ejecución*

*// Error: Sólo se puede acceder a un miembro const enum utilizando un literal de cadena.*

```
// console.log(NinjaActivity[miActividadNinjaFavorita]);
```

A modo de comparación, un Enum normal

```
// TypeScript: Un Enum normal
enum PirateActivity {
    Abordar, Beber,
    Esgrima
}

// JavaScript: El Enum después de la compilación
// var PirateActivity;
// (function (PirateActivity) {
//     PirateActivity[PirateActivity["Boarding"]= 0] = "Abordaje"
//     PirateActivity[PirateActivity["Drinking"]= 1]= "Drinking";
//     PirateActivity[PirateActivity["Esgrima"]= 2]= "Esgrima";
// })(PirateActivity|| (PirateActivity= {}));

// TypeScript: Un uso normal de este Enum
deje miActividadPirataFavorita= PirateActivity.Boarding;
console.log(miActividadPirataFavorita); // 0

// JavaScript: Parece bastante similar en JavaScript
// var myFavoritePirateActivity= PirateActivity.Boarding;
// console.log(miActividadPirataFavorita); // 0

// TypeScript: Y algún otro uso normal
console.log(PirateActivity["Beber"]); // 1

// JavaScript: Parece bastante similar en JavaScript
// console.log(PirateActivity["Drinking"]); // 1

// TypeScript: En tiempo de ejecución, puede acceder a un enum normal
console.log(PirateActivity[miActividadPirataFavorita]); // "Abordando"

// JavaScript: Y se resolverá en tiempo de ejecución
// console.log(PirateActivity[miActividadPirataFavorita]); // "Abordando"
```

## Sección 3.8: Número

Al igual que en JavaScript, los números son valores de coma flotante.

```
let pi: n úmero= 3.14; // base 10 decimal por defecto
let hexadecimal: n úmero= 0xFF; // 255 en decimal
```

ECMAScript 2015 permite binario y octal.

```
let binario: n úmero= 0b10; // 2 en decimal
let octal: n úmero= 0o755; // 493 en decimal
```

## Sección 3.9: Cadena

Tipo de dato textual:

```
let comillas simples: cadena= 'simple';
let comillas dobles: cadena= "doble";
let templateString: string= `Soy$ { singleQuotes }` ; // Soy single
```

## Sección 3.10: Matriz

Una matriz de valores:

```
let tresCerdos: number[] = [1, 2, 3];  
let genericStringArray: Array< string > = ['first', '2nd', '3rd'];
```

## Sección 3.11: Enum

Un tipo para nombrar un conjunto de valores

numéricos: Los valores numéricos por

defecto son 0:

```
enum Día { Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo };  
let mejorDía: Day = Day.Saturday;
```

Establece un número inicial por defecto:

```
enum TenPlus { Diez = 10, Once, Doce }
```

o asignar valores:

```
enum MyOddSet { Tres = 3, Cinco = 5, Siete = 7, Nueve = 9 }
```

## Sección 3.12: Cualquier

Si no está seguro del tipo, puede elegir cualquiera :

```
let cualquier cosa: cualquier = 'Soy una  
cadena'; cualquier cosa = 5; // pero ahora soy  
el número 5
```

## Sección 3.13: Nulo

Si no tiene ningún tipo, comúnmente utilizado para funciones que no devuelven nada:

```
function log(): void { console.log('No  
devuelvo nada');  
}
```

Tipos **void** Sólo se pueden asignar **nulos** o **indefinidos**.

# Capítulo 4: Matrices

## Sección 4.1: Encontrar un objeto en una matriz

Uso de find()

```
const inventario= [
  {nombre: 'manzanas', cantidad: 2},
  {nombre: 'plátanos', cantidad: 0},
  {nombre: 'cerezas', cantidad: 5}
];

function encontrarFrutas(fruta) {
  return fruta.nombre === 'cerezas';
}

inventory.find(findCherries);      // { nombre: 'cerezas', cantidad: 5 }

/* 0 */

inventory.find(e=> e.name === 'manzanas');  // { name: 'manzanas', quantity: 2 }
```

# Capítulo 5: Enums

## Sección 5.1: Enums con valores explícitos

Por defecto todos los valores `enum` se resuelven a números. Digamos que si tiene algo como

```
enum MimeType {  
  JPEG,  
  PNG,  
  PDF  
}
```

el valor real detrás de, por ejemplo, `MimeType.PDF` será 2.

Pero algunas veces es importante que el enum resuelva a un tipo diferente. Por ejemplo, usted recibe el valor de backend / frontend / otro sistema que es definitivamente una cadena. Esto podría ser un dolor, pero por suerte existe este método:

```
enum MimeType {  
  JPEG = < cualquiera>  
    'image/jpeg', PNG = <  
    cualquiera> 'image/png',  
  PDF = < cualquier> 'application/pdf'  
}
```

Esto resuelve el `MimeType.PDF` a `application/pdf`. Desde

TypeScript 2.4 es posible declarar [enums de cadena](#):

```
enum MimeType {  
  JPEG= 'image/jpeg',  
  PNG= 'image/png',  
  PDF= 'application/pdf',  
}
```

Puede proporcionar explícitamente valores numéricos utilizando el mismo método

```
enum MyType {  
  Valor= 3,  
  ValueEx= 30,  
  ValorEx2= 300  
}
```

Los tipos más sofisticados también funcionan, ya que los enums no-const son objetos reales en tiempo de ejecución, por ejemplo

```
enum FancyType { OneArr  
  =< any>[1],  
  TwoArr= < any>[2, 2],  
  ThreeArr =< any>[3, 3, 3]  
}
```

se convierte en

```
var FancyType; ( function  
(FancyType) {  
  FancyType[FancyType["OneArr"] = [1]] = "OneArr";  
  FancyType[FancyType["TwoArr"]= [2, 2]] = "DosArr";
```

```
FancyType[FancyType["ThreeArr"]= [3, 3, 3]] = "ThreeArr"
})(FancyType| (FancyType= {}));
```

## Sección 5.2: Cómo obtener todos los valores del enum

```
enum CiertaEnum { A, B }

let enumValues:Array<cadena>= [];

for( let valor in CiertaEnum) {
    if( typeof SomeEnum[value]=== 'number') {
        enumValues.push(value);
    }
}

enumValues.forEach(v=> console.log(v))
//A
//B
```

## Sección 5.3: Ampliación de sumas sin implementación de sumas personalizadas

```
enum FuenteEnum {
    valor1 =< cualquiera>
    'valor1', valor2= <
    cualquiera> 'valor2'
}

enum AdditionToSourceEnum {
    value3= < any> 'value3',
    value4= < any> 'value4'
}

// necesitamos este tipo para que TypeScript resuelva los tipos correctamente
tipo TestEnumType= SourceEnum| AdditionToSourceEnum;
// y necesitamos este valor "instancia" para utilizar valores
let TestEnum= Object.assign({}, SourceEnum, AdditionToSourceEnum);
// también funciona bien la función TypeScript 2
// let TestEnum= { ...SourceEnum, ...AdditionToSourceEnum };

function check(test: TestEnumType) {
    return test=== TestEnum.value2;
}

console.log(TestEnum.value1); console.log(TestEnum.value2
===< any> 'value2'); console.log(check(TestEnum.value2));
console.log(check(TestEnum.value3));
```

## Sección 5.4: Implementación de enum personalizados: extiende para enums

A veces es necesario implementar Enum por su cuenta. Por ejemplo, no hay una forma clara de extender otros enums. La implementación personalizada permite esto:

```
class Enum {
    constructor(valor protegido: cadena) {}

    public toString() {
        return String( este.valor);
    }
}
```



```

    }

    public is(valor: Enum | cadena) {
        return this.value === value.toString();
    }
}

class FuenteEnum extends Enum {
    public static valor1 = new SourceEnum('valor1'); public
    static valor2 = new SourceEnum('valor2');
}

class TestEnum extends SourceEnum {
    public static valor3 = new TestEnum('valor3'); public
    static valor4 = new TestEnum('valor4');
}

function check(test: TestEnum) {
    return test === TestEnum.value2;
}

let valor1 = TestEnum.valor1;

console.log(valor1 + 'hola');
console.log(valor1.toString() === 'valor1');
console.log(valor1.is('valor1'));
console.log(!TestEnum.valor3.is(TestEnum.valor3));
console.log(check(TestEnum.valor2));
// esto funciona pero quizás su TSLint se quejaría
// ¡atención! no funciona ===
// utilice .is() en su lugar
console.log(TestEnum.value1 ===< any> 'value1');

```

# Capítulo 6: Funciones

## Sección 6.1: Parámetros opcionales y por defecto

### Parámetros opcionales

En TypeScript, se asume que cada parámetro es requerido por la función. Puedes añadir una `?` al final del nombre de un parámetro para establecerlo como opcional.

Por ejemplo, el parámetro `lastName` de esta función es opcional:

```
function buildName(firstName: string, lastName?: string) {  
    // ...  
}
```

Los parámetros opcionales deben ir después de todos los parámetros no opcionales:

```
function buildName(firstName?: string, lastName: string) // No válido
```

### Parámetros por defecto

Si el usuario pasa `undefined` o no especifica un argumento, se asignará el valor por defecto. Se denominan parámetros *inicializados por defecto*.

Por ejemplo, "Smith" es el valor por defecto del parámetro `lastName`.

```
function buildName(firstName: cadena, lastName= "Smith") {  
    // ...  
}  
  
buildName('foo', 'bar');           // firstName== 'foo', lastName== 'bar'  
buildName('foo');                 // firstName== 'foo', lastName== 'Smith'  
buildName('foo', undefined);      // firstName== 'foo', lastName== 'Smith'
```

## Sección 6.2: Función como parámetro

Supongamos que queremos recibir una función como parámetro, podemos hacerlo así:

```
function foo(otherFunc: Function): void {  
    ...  
}
```

Si queremos recibir un constructor como parámetro:

```
function foo(constructorFunc: { new(): any }) {  
    new constructorFunc();  
}  
  
function foo(constructorWithParamsFunc: { new(num: number): any }) {  
    nuevo constructorWithParamsFunc(1);  
}
```

O para hacerlo más fácil de leer podemos definir una interfaz que describa el constructor:

```
interface IConstructor {  
    new();  
}
```

```

}

function foo(constructorFunc: IConstructor) {
    new constructorFunc();
}

```

O con parámetros:

```

interfaz INumberConstructor {
    nuevo(núm: número);
}

function foo(constructorFunc: INumberConstructor) {
    new constructorFunc(1);
}

```

Incluso con genéricos:

```

interfaz ITConstructor< T, U> {
    new(item: T): U;
}

function foo< T, U>(constructorFunc: ITConstructor< T, U>, item: T): U {
    return new constructorFunc(item);
}

```

Si queremos recibir una función simple y no un constructor es casi lo mismo:

```

function foo(func: { (): void }) {
    func();
}

function foo(constructorConParámetrosFunc: { (num: número): void }) {
    nuevo constructorWithParamsFunc(1);
}

```

O, para facilitar la lectura, podemos definir una interfaz que describa la función:

```

interfaz IFunction {
    (): void;
}

function foo(func: IFunction ) {
    func();
}

```

O con parámetros:

```

interfaz INumberFunction {
    (num: número): cadena;
}

function foo(func: INumberFunction ) {
    func(1);
}

```

Incluso con genéricos:

```

interfaz ITFunc< T, U> {
    (item: T): U;
}

function foo< T, U>(constructorFunc: ITFunc< T, U>, item: T): U {
    return func(item);
}

```

## Sección 6.3: Funciones con tipos de unión

Una función TypeScript puede recibir parámetros de múltiples tipos predefinidos utilizando tipos de unión.

```

function whatTime(hora:n número| cadena, minuto:n número| cadena):cadena{
    return hora+ ':'+ minuto;
}

```

```

whatTime(1,30)           //'1:30'
whatTime('1',30)         //'1:30'
whatTime(1,'30')         //'1:30'
whatTime('1','30')       //'1:30'

```

TypeScript trata estos parámetros como un único tipo que es una unión de los otros tipos, por lo que tu función debe ser capaz de manejar parámetros de cualquier tipo que esté en la unión.

```

function addTen(inicio:n número| cadena):n número{
    if( typeof number=== 'string'){
        return parseInt(n número)+10;
    }else{
        si no, devuelve el n número+ 10;
    }
}

```

## Sección 6.4: Tipos de funciones

Funciones con nombre

```

function multiplicar(a, b) {
    devuelve a * b;
}

```

Funciones anónimas

```

let multiply= function(a, b) { return a * b; };

```

Funciones lambda / flecha

```

let multiplicar= (a, b)=> { return a * b; };

```

# Capítulo 7: Clases

TypeScript, al igual que ECMAScript 6, admite la programación orientada a objetos mediante clases. Esto contrasta con las versiones anteriores de JavaScript, que solo admitían la cadena de herencia basada en prototipos.

El soporte de clases en TypeScript es similar al de lenguajes como Java y C#, en el sentido de que las clases pueden heredar de otras clases, mientras que los objetos se instancian como instancias de clase.

También de forma similar a esos lenguajes, las clases TypeScript pueden implementar interfaces o hacer uso de genéricos.

## Sección 7.1: Clases abstractas

```
class abstracta M {
  constructor(public fabricante: cadena) {
  }

  // Una clase abstracta puede definir métodos propios o...
  resumen(): cadena {
    return `${este.fabricante} fabrica esta máquina.`;
  }

  // Requerir a las clases herederas que implementen métodos
  abstracto masInfo(): cadena;
}

class Coche extends M {
  constructor(fabricante: cadena, public posicion: n, public velocidad: n, public velocidadMaxima: n) {
    super(fabricante);
  }

  move() {
    this.posicion += this.velocidad;
  }

  masInfo() {
    return `¡Este es un coche situado en ${this.posicion} y que va a ${this.velocidad}mph!`;
  }
}

let myCar = new Coche("Konda", 10, 70);
myCar.move(); // la posición es ahora 80
console.log(myCar.resumen()); // imprime "Konda fabrica esta máquina".
console.log(myCar.masInfo()); // imprime "¡Este es un coche situado a 80 y que va a 70mph!"
```

Las clases abstractas son clases base a partir de las cuales pueden extenderse otras clases. No se pueden instanciar (por ejemplo, no se puede hacer `new Machine("Konda")`).

Las dos características clave de una clase abstracta en TypeScript son:

1. Pueden aplicar métodos.
2. Pueden definir métodos que las clases herederas deben implementar.

Por este motivo, las clases abstractas pueden considerarse conceptualmente una combinación de una interfaz y una clase.

## Sección 7.2: Clase simple

```
class Coche {
```

```

posición pública: número= 0;
velocidad privada: número= 42;

move() {
    this.position+= this.speed;
}
}

```

En este ejemplo, declaramos una clase simple `Coche`. La clase tiene tres miembros: una propiedad privada `speed`, una propiedad pública `position` y un método público `move`. Observa que cada miembro es público por defecto. Por eso `move()` es público, aunque no hayamos utilizado la palabra clave `public`.

```

var car= new Car();           // crear una instancia de
Coche car.move();             // llamar a un método
console.log(car.position);     // acceder a una propiedad
pública

```

## Sección 7.3: Herencia básica

```

class Coche {
    public posición: número= 0;
    protected velocidad: número= 42;

    move() {
        this.position+= this.speed;
    }
}

class SelfDrivingCar extends Coche {

    move() {
        // empieza a moverte :->
        super.move();
        super.move();
    }
}

```

Este ejemplo muestra cómo crear una subclase muy simple de la clase `Car` utilizando la palabra clave `extends`. La página [Herencia en TypeScript](#) explica la herencia en TypeScript. La clase `SelfDrivingCar` anula el método `move()` y utiliza la implementación de la clase base mediante `super`.

## Sección 7.4: Constructores

En este ejemplo utilizamos el constructor para declarar una propiedad pública `position` y una propiedad protegida `speed` en la clase base. Estas propiedades se denominan *propiedades parámetro*. Nos permiten declarar un parámetro del constructor y un miembro en un solo lugar.

Una de las mejores cosas en TypeScript, es la asignación automática de los parámetros del constructor a la propiedad correspondiente.

```

class Coche {
    public posición: número;
    protected velocidad:
    número;

    constructor(posición: número, velocidad: número) {
        this.position= position;
        this.speed= speed;
    }

    move() {
        this.position+= this.speed;
    }
}

```

```
}
}
```

Todo este código puede resumirse en un único constructor:

```
class Coche {
  constructor(public posición: number, protected velocidad: number) {}

  move() {
    this.position += this.speed;
  }
}
```

Y ambos serán transpilados de TypeScript (tiempo de diseño y tiempo de compilación) a JavaScript con el mismo resultado, pero escribiendo significativamente menos código:

```
var Car= ( function () {
  function Coche(posición, velocidad) {
    this.posición= posición;
    this.velocidad= velocidad;
  }
  Car.prototype.move= function () {
    this.position += this.speed;
  };
  coche de vuelta;
})();
```

Los constructores de las clases derivadas tienen que llamar al constructor de la clase base con `super()`.

```
class SelfDrivingCar extends Car {
  constructor(startAutoPilot: boolean) {
    super(0, 42);
    si (startAutoPilot) {
      this.move();
    }
  }
}

let car= new SelfDrivingCar( true);
console.log(car.position); // acceder a la propiedad pública position
```

## Sección 7.5: Accesores

En este , modificamos el ejemplo "Clase simple" para permitir el acceso a la propiedad `speed`. Los `accessors` de TypeScript nos permiten añadir código adicional en `getters` o `setters`.

```
class Coche {
  public posición: number= 0;
  private _velocidad: number= 42; private _MAX_SPEED= 100

  move() {
    this.position += this._speed;
  }

  obtener velocidad(): number {
    return this._velocidad;
  }
}
```

```

    set velocidad(valor: n    úmero) {
        this._speed= Math.min(value,    this._MAX_SPEED);
    }
}

let car= new Car();
car.speed= 120;
console.log(velocidad.coche);           // 100

```

## Sección 7.6: Transpilación

Dada una clase SomeClass, veamos cómo se transpila TypeScript a JavaScript.

Fuente TypeScript

```

class CiertaClase {

    public static SomeStaticValue: cadena= "hola";
    public someMemberValue: n    úmero= 15;
    private somePrivateValue: boolean=    false;

    constructor () {
        SomeClass.SomeStaticValue= SomeClass.getGoodbye();
        this.someMemberValue= this.getFortyTwo();
        this.somePrivateValue=    this.getTrue();
    }

    public static getAdi    ós(): string {
        devuelve "¡adi    ós!";
    }

    public getFortyTwo(): n    úmero {
        devolver 42;
    }

    private getTrue(): boolean {
        devuelve true;
    }

}

```

Fuente JavaScript

Cuando se transpila usando TypeScript v2.2.2 , la salida es así:

```

var SomeClass= (function () {
    function SomeClass() {
        this.someMemberValue= 15;
        this.somePrivateValue=    false;
        SomeClass.SomeStaticValue= SomeClass.getGoodbye();
        this.someMemberValue= this.getFortyTwo();
        this.somePrivateValue=    this.getTrue();
    }
    CiertaClase.getAdi    ós= function () {
        devuelve "¡adi    ós!";
    };
    CiertaClase.    prototype.getCuarentaDos= function () {
        devolver 42;
    };
    CiertaClase.    prototype.getTrue= function () {
        devuelve true;
    };
}

```



```
    return AlgunaClase;
})();
SomeClass.SomeStaticValue= "hola";
```

#### Observaciones

- La modificación del prototipo de la clase se envuelve dentro de un [IIFE](#).
- Las variables miembro se definen dentro de la **función** principal de la clase.
- Las propiedades estáticas se añaden directamente al objeto de clase, mientras que las propiedades de instancia se añaden a la clase prototipo.

## Sección 7.7: Monkey patch una función en una clase existente

A veces es útil poder extender una clase con nuevas funciones. Por ejemplo, supongamos que una cadena debe ser convertida a una cadena camel case. Entonces necesitamos decirle a TypeScript, que String contiene una función llamada toUpperCase, que devuelve una cadena.

```
interfaz String {
    toUpperCase(): string;
}
```

Ahora podemos parchear esta función en la implementación de String .

```
String.prototype.toUpperCase= function() : string {
    return this.replace( /[^\w-]/g, '')
        .replace( /(?![\w-][A-Z])[\w-]+/g (match: any, index: number)=> {
            return+ match=== 0 ? "" : coincidencia[ índice=== 0 ? 'toLowerCase' : 'toUpperCase']();
        });
}
```

Si se carga esta extensión de String , se puede utilizar así:

```
"Esto es un ejemplo".toUpperCase(); //=> "thisIsAnExample"
```

# Capítulo 8: Decorador de clases

Parámetro

Detalles

objetivo

La clase que se va a decorar

## Apartado 8.1: Generación de metadatos mediante un decorador de clases

Esta vez vamos a declarar un decorador de clase que añadirá algunos metadatos a una clase cuando le apliquemos:

```
function addMetadata(target: any) {  
  
    // Añadir algunos metadatos  
    target.customMetadata = {  
        someKey: "alg únValor"  
    };  
  
    // Objetivo de retorno  
    objetivo de retorno;  
  
}
```

A continuación, podemos aplicar el decorador de clase:

```
@addMetadata  
class Persona {  
    privado _nombre: cadena;  
    public constructor(nombre: cadena) {  
        this._name = nombre;  
    }  
    public saludar() {  
        devolver este._nombre;  
    }  
}  
  
function getMetadataFromClass(target: any) {  
    return target.customMetadata;  
}  
  
console.log(getMetadataFromClass(Person));
```

El decorador se aplica cuando se declara la clase, no cuando creamos instancias de la clase. Esto significa que los metadatos se comparten entre todas las instancias de una clase:

```
function getMetadataFromInstance(target: any) {  
    return target.constructor.customMetadata;  
}  
  
let persona1 = new Person("Juan");  
let persona2 = new Person("Lisa");  
  
console.log(getMetadataFromInstance(persona1));  
console.log(getMetadataFromInstance(persona2));
```

## Sección 8.2: Pasar argumentos a un decorador de clase

Podemos envolver un decorador de clase con otra función para permitir la personalización:

```
function addMetadata(metadata: any) {
    return function log(target: any) {

        // Añadir metadatos
        target.customMetadata= metadata;

        // Objetivo de retorno
        objetivo de    retorno;

    }
}
```

El addMetadata toma algunos argumentos utilizados como configuración y luego devuelve una función sin nombre que es el decorador real. En el decorador podemos acceder a los argumentos porque hay un cierre en su lugar.

A continuación, podemos invocar el decorador pasando algunos valores de configuración:

```
@addMetadata({ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" })
class Persona {
    privado _nombre: cadena;
    public constructor(nombre: cadena) {
        this._name= nombre;
    }
    public saludar() {
        devolver este._nombre;
    }
}
```

Podemos utilizar la siguiente función para acceder a los metadatos generados:

```
function getMetadataFromClass(target: any) {
    return target.customMetadata;
}

console.log(getMetadataFromInstance(Persona));
```

Si todo ha ido bien debería aparecer la consola:

```
{ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" }
```

## Sección 8.3: Decorador básico de clases

Un decorador de clase no es más que una función que toma la clase como único argumento y la devuelve después de hacer algo con ella:

```
function log< T>(objetivo: T) {

    // Hacer algo con el objetivo
    console.log(objetivo);

    // Objetivo de retorno
    objetivo de    retorno;

}
```

A continuación, podemos aplicar el decorador de clase a una clase:

@log

```
clase Persona {  
  privado _nombre: cadena;  
  public constructor(nombre: cadena) {  
    this._name= nombre;  
  }  
  public saludar() {  
    devolver este._nombre;  
  }  
}
```

# Capítulo 9: Interfaces

Una interfaz especifica una lista de campos y funciones que pueden esperarse de cualquier clase que implemente la interfaz. A la inversa, una clase no puede implementar una interfaz a menos que tenga todos los campos y funciones especificados en la .

La principal ventaja del uso de interfaces es que permite utilizar objetos de distintos tipos de polimórfica. Esto se debe a que cualquier clase que implemente la interfaz tiene al menos esos campos y funciones.

## Sección 9.1: Interfaz extensible

Supongamos que tenemos una interfaz:

```
interfaz IPerson {  
    nombre: cadena;  
    edad: n úmero;  
  
    aliento(): void;  
}
```

Y queremos crear una interfaz más específica que tenga las mismas propiedades de la persona, podemos hacerlo utilizando la función [extiende la](#) palabra clave:

```
interface IManager extends IPerson {  
    managerId: n úmero;  
  
    managePeople(people: IPerson[]): void;  
}
```

Además, es posible ampliar varias interfaces.

## Sección 9.2: Interfaz de clase

Declare variables públicas y métodos tipo en la interfaz para definir cómo otro código typescript puede interactuar con ella.

```
interfaz ISampleClassInterface {  
    sampleVariable: string;  
  
    sampleMethod(): void;  
  
    optionalVariable?: string;  
}
```

Aquí creamos una clase que implementa la interfaz.

```
class SampleClass implements ISampleClassInterface {  
    public sampleVariable: string;  
    privado answerToLifeTheUniverseAndEverything: n úmero;  
  
    constructor() {  
        this.sampleVariable = 'valor de cadena';  
        this.answerToLifeTheUniverseAndEverything = 42;  
    }  
  
    public sampleMethod(): void {  
        // no hacer nada  
    }  
}
```

```
private respuesta(q: cualquiera): número {
    return this.answerVidaElUniversoYTodo;
}
}
```

El ejemplo muestra cómo crear una interfaz `ISampleClassInterface` y una clase `SampleClass` que implemente la interfaz.

## Sección 9.3: Uso de interfaces para polimorfismo

La razón principal para usar interfaces es conseguir polimorfismo y proporcionar a los desarrolladores la posibilidad de implementar a su manera en el futuro los métodos de la interfaz.

Supongamos que tenemos una interfaz y tres clases:

```
interface Conector{
    doConnect(): boolean;
}
```

Esta es la interfaz del conector. Ahora vamos a implementar que para la comunicación Wifi.

```
export class WifiConnector implements Conector{

    public doConnect(): boolean{
        console.log("Conectando por wifi");
        console.log("Obtener contraseña");
        console.log("Alquilar una IP durante 24 horas");
        console.log("Conectado");
        devolver true
    }

}
```

Aquí hemos desarrollado nuestra clase concreta llamada `WifiConnector` que tiene su propia implementación. Esta es ahora la clase `Conector`.

Ahora estamos creando nuestro `Sistema` que tiene un componente `Conector`. Esto se llama inyección de dependencia.

```
export class Sistema {
    constructor(private conector: Conector){ #inyectar conector tipo conector.doConnect()
    }
}
```

`constructor(private conector: Conector)` esta línea es muy importante aquí. `Conector` es una interfaz y debe tener `doConnect()`. Como `Conector` es una interfaz esta clase `System` tiene mucha más flexibilidad. Podemos pasar cualquier `Type` que tenga implementada la interfaz `Conector`. En el futuro los desarrolladores tendrán mas flexibilidad. Por ejemplo, ahora el desarrollador quiere añadir un módulo de conexión Bluetooth:

```
export class ConectorBluetooth implements Conector{

    public doConnect(): boolean{
        console.log("Conectando por Bluetooth");
        console.log("Emparejar con PIN");
        console.log("Conectado");
        devolver true
    }

}
```

```
}
```

Ver que Wifi y Bluetooth tienen su propia implementación. Su propia manera diferente de conectarse. Sin embargo, ambos han implementado `Type Connector` y ahora son `Type Connector`. De modo que podemos pasar cualquiera de ellos a la clase `System` como parámetro del constructor. Esto se llama polimorfismo. La clase `System` ahora no es consciente de si es Bluetooth / Wifi, incluso podemos añadir otro módulo de comunicación como infrarrojos, Bluetooth5 y lo que sea con sólo implementar la interfaz `Connector`.

Esto se llama [Duck typing](#). El tipo de conector es ahora dinámico ya que `doConnect()` es sólo un marcador de posición y el desarrollador lo implementa como propio.

si en `constructor(private connector: WifiConnector)` donde `WifiConnector` es una clase concreta ¿qué pasará? Entonces la clase `System` se acoplará estrechamente sólo con `WifiConnector` nada más. Aquí interfaz resuelto nuestro problema por polimorfismo.

## Sección 9.4: Interfaces genéricas

Al igual que las clases, las interfaces también pueden recibir parámetros polimórficos (también conocidos como genéricos).

Declaración de parámetros genéricos en interfaces

```
interfaz IStatus U<> {  
    código: U;  
}  
  
interfaz IEvents< T> {  
    list: T[];  
    emit(evento: T): void;  
    getAll(): T[];  
}
```

Aquí, puedes ver que nuestras dos interfaces toman algunos parámetros genéricos, `T` y `U`.

Implementación de Interfaces Genéricas

Crearemos una clase sencilla para implementar la interfaz `IEvents`.

```
class Estado T<> implements IEventos T<> {  
  
    lista: T[];  
  
    constructor() {  
        this.list = [];  
    }  
  
    emit(evento: T): void {  
        this.list.push(evento);  
    }  
  
    getAll(): T[] {  
        devolver esta.lista;  
    }  
}
```

Vamos a crear algunas instancias de nuestra clase `State`.

En nuestro ejemplo, la clase `State` manejará un estado genérico utilizando `IStatus T<>`. De este modo, la interfaz

IEvent T.<> también gestionará un IStatus T.<>

```
const s = new State< IStatus< number>>();

// Se espera que la propiedad 'code' sea un número, así:
s.emit({ code: 200 }); // funciona
s.emit({ code: '500' }); // error de tipo

s.getAll().forEach(evento=> console.log(evento.code));
```

Aquí nuestra clase State se escribe como IStatus< número .>

```
const s2 = new State< IStatus< Code>>();

//Podemos emitir código como el tipo Código
s2.emit({ código: { mensaje: 'OK', estado: 200 } });

s2.getAll().map(event=> event.code).forEach(event=> {
  console.log(event.message); console.log(event.status);
});
```

Nuestra clase State está tipada como IStatus< Code> . De esta , podemos pasar tipos más complejos a nuestro método emit.

Como puedes ver, las interfaces genéricas pueden ser una herramienta muy útil para el código tipado estáticamente.

## Sección 9.5: Añadir funciones o propiedades a una interfaz existente

Supongamos que tenemos una referencia a la definición de tipo de JQuery y queremos extenderla para tener funciones adicionales de un plugin que incluimos y que no tiene una definición de tipo oficial. Podemos extenderla fácilmente declarando las funciones añadidas por el plugin en una declaración de interfaz separada con el mismo nombre de JQuery :

```
interfaz JQuery {
  pluginFunctionQueNoHaceNada(): void;

  // crear función encadenable
  manipulateDOM(HTMLElement): JQuery;
}
```

El compilador fusionará todas las declaraciones con el mismo en una sola - ver [fusión de declaraciones](#) para más detalles.

## Sección 9.6: Implementación implícita y forma del objeto

TypeScript soporta interfaces, pero el compilador genera JavaScript, que no las soporta. Por lo tanto, las interfaces se pierden en el paso de compilación. Esta es la razón por la que la comprobación de tipos en interfaces se basa en *la forma* del objeto -es decir, si el objeto soporta los campos y funciones de la interfaz- y no en si la interfaz está realmente implementada o no.

```
interfaz IKickable { kick(distancia:
  número): void;
}
clase Bola {
  kick(distancia: número): void {
    console.log(" ¡Pateado", distancia, "metros!");
  }
}
```



```

}
let kickable: IKickable = new Ball();
kickable.kick(40);

```

Por lo tanto, aunque `Ball` no implemente explícitamente `IKickable`, una instancia de `Ball` puede asignarse a (y manipularse como) una instancia de `IKickable`, incluso cuando se especifica el tipo.

## Sección 9.7: Uso de interfaces para aplicar tipos

Uno de los principales beneficios de TypeScript es que hace cumplir los tipos de datos de los valores que usted está pasando alrededor de su código para ayudar a prevenir errores.

Digamos que estás haciendo una aplicación de citas para mascotas.

Tienes esta sencilla función que comprueba si dos mascotas son compatibles entre sí...

```

checkCompatible(mascotaUno, mascotaDos) {
  if (petOne.species === petTwo.species &&
    Math.abs(petOne.age - petTwo.age) <= 5) {
    devuelve true;
  }
}

```

Este es un código completamente funcional, pero sería demasiado fácil para alguien, especialmente para otras personas que trabajan en esta aplicación y que no escribieron esta función, no ser conscientes de que se supone que deben pasarle objetos con las propiedades 'especie' y 'edad'. Podrían probar erróneamente `checkCompatible(petOne.species, petTwo.species)` y luego tener que averiguar los errores que se producen cuando la función intenta acceder a `petOne.species.species` o `petOne.species.age`.

Una forma de evitar que esto ocurra es especificar las propiedades que queremos en los parámetros de las mascotas:

```

checkCompatible(petOne: {especie: cadena, edad: n    úmero}, petTwo: {especie: cadena, edad: n    úmero}) {
  //...
}

```

En este, TypeScript se asegurará de que todo lo que se pase a la función tenga las propiedades 'species' y 'age' (no pasa nada si tienen propiedades adicionales), pero esta es una solución un poco difícil de manejar, incluso con sólo dos propiedades especificadas. Con interfaces, ¡hay una forma mejor!

Primero definimos nuestra interfaz:

```

interfaz Mascota {
  especie: cadena;
  edad: n    úmero;
  //Podemos añadir más propiedades si lo deseamos.
}

```

Ahora todo lo que tenemos que hacer es especificar el tipo de nuestros parámetros como nuestra nueva interfaz, así...

```

checkCompatible(petOne: Mascota, petTwo: Mascota) {
  //...
}

```

... ¡y TypeScript se asegurará de que los parámetros pasados a nuestra función contengan las propiedades especificadas en la interfaz `Pet`!

# Capítulo 10: Genéricos

## Sección 10.1: Interfaces genéricas

Declarar una interfaz genérica

```
interfaz IResult< T> {  
    wasSuccessful: boolean;  
    error: T;  
}  
  
var resultado: IResult< cadena >=...  
var error: string= result.error;
```

Interfaz genérica con múltiples parámetros de tipo

```
interfaz IRunnable< T, U> {  
    run(input: T): U;  
}  
  
var runnable: IRunnable< cadena, número >=...  
var input: cadena;  
var resultado: número= runnable.run(input);
```

Implementación de una interfaz genérica

```
interfaz IResult< T>{  
    wasSuccessful: boolean;  
    error: T;  
  
    clone(): IResultado< T>;  
}
```

Implementalo con una clase genérica:

```
class Resultado T<> implements IResult T<> {  
    constructor(public resultado: booleano, public error: T) {  
    }  
  
    public clone(): IResult T<> {  
        return new Resultado< T>( este.resultado,     este.error);  
    }  
}
```

Implementalo con una clase no genérica:

```
class StringResult implements IResult< string> {  
    constructor(public result: boolean, public error: string) {  
    }  
  
    public clone(): IResult< string> {  
        return new StringResult( this.result,     this.error);  
    }  
}
```

## Sección 10.2: Clase genérica

```
class Resultado T<> {
```

```

    constructor(public tuvoExito: boolean, public error: T) {
    }

    public clone(): Resultado T<> {
        ...
    }
}

let r1= new Result( false, 'error: 42');           // El compilador infiere T a
cadena let r2= new Result( false, 42);           // El compilador infiere T a número
let r3= new Result< string>( true, null);           // Establece explícitamente T
a cadena
let r4= new Resultado< cadena>( true, 4);           // Error de compilación porque 4 no es una cadena

```

## Sección 10.3: Parámetros de tipo como restricciones

Con TypeScript 1.8 es posible que una restricción de parámetro de tipo haga referencia a parámetros de tipo de la misma lista de parámetros de tipo. Anteriormente esto era un error.

```

function assign< T extends U, U>(target: T, source: U): T {
    for ( let id in source) {
        target[id]= source[id];
    }
    objetivo de retorno;
}

let x= { a: 1, b: 2, c: 3, d: 4 };
assign(x, { b: 10, d: 20 });
assign(x, { e: 0 });           // Error

```

## Sección 10.4: Restricciones genéricas

Restricción simple:

```

interfaz IRunnable {
    run(): void;
}

interfaz IRunner< T extends IRunnable> {
    runSafe(runnable: T): void;
}

```

Restricción más compleja:

```

interfaz IRunnable U<> { run():
    U;
}

interfaz IRunner< T extends IRunnable< U>, U> {
    runSafe(runnable: T): U;
}

```

Aún más complejo:

```

interfaz IRunnable< V> {
    run(par ámetro: U): V;
}

interfaz IRunner< T extends IRunnable< U, V>, U, V> {

```

```
runSafe(ejecutable: T, parámetro: U): V;
}
```

Restricciones de tipo inline:

```
interfaz IRunnable< T extends { run(): void }> { runSafe(runnable:
T): void;
}
```

## Sección 10.5: Funciones genéricas

En interfaces:

```
interfaz IRunner {
runSafe< T extends IRunnable>(runnable: T): void;
}
```

En las clases:

```
class Runner implements IRunner {

public runSafe< T extends IRunnable>(runnable: T): void {
    intentar {
        runnable.run();
    } catch(e) {
    }
}

}
```

Funciones sencillas:

```
function runSafe< T extends IRunnable>(runnable: T): void {
    intentar {
        runnable.run();
    } catch(e) {
    }
}
```

## Sección 10.6: Uso de clases y funciones genéricas:

Crear instancia de clase genérica:

```
var stringRunnable= new Runnable< string>();
```

Ejecutar función genérica:

```
function runSafe< T extends Runnable< U>, U>(runnable: T);
```

*// Especificar los tipos genéricos:*

```
runSafe< Runnable< cadena>, cadena>(cadenaRunnable);
```

*// Deja que typescript calcule los tipos genéricos por sí mismo:*

```
runSafe(stringRunnable);
```

# Capítulo 11: Comprobación estricta de nulos

## Sección 11.1: Comprobación estricta de nulos en acción

Por defecto, todos los tipos en TypeScript permiten `null`:

```
function getId(x: Elemento) {  
    devolver x.id;  
}  
getId( null); // TypeScript no se queja, pero esto es un error de ejecución.
```

TypeScript 2.0 añade soporte para comprobaciones estrictas de nulos. Si establece `--strictNullChecks` al ejecutar `tsc` (o establece esta opción en `tsconfig.json`), los tipos ya no permitirán **nulos**:

```
function getId(x: Elemento) {  
    devolver x.id;  
}  
getId( null); // error: Argumento de tipo 'null' no es assignable a parámetro de tipo 'Element'.
```

Debe permitir valores **nulos** explícitamente:

```
function getId(x: Elemento | null) {  
    return x.id; // error TS2531: Object is possibly 'null'.  
}  
getId( null);
```

Con una protección adecuada, el tipo de código se comprueba y se ejecuta correctamente:

```
function getId(x: Elemento | null) {  
    si (x) {  
        return x.id; // En esta rama, el tipo de x es Elemento  
    } else {  
        return null; // En esta rama, el tipo de x es null.  
    }  
}  
getId( null);
```

## Sección 11.2: Afirmaciones no nulas

El operador de aserción no nulo, `!`, permite afirmar que una expresión no es **nula** o **indefinida** cuando el compilador de TypeScript no puede inferirlo automáticamente:

```
type ListNode = { datos: n    úmero; next?: ListNode; };  
  
function addNext(node: ListNode) {  
    if (node.next === undefined) { node.next  
        = {data: 0};  
    }  
}  
  
function setNextValue(node: ListNode, value: number) {  
    addNext(node);  
  
    // Aunque sabemos que `node.next` está definido porque acabamos de llamar a `addNext`,  
    // TypeScript no es capaz de inferir esto en la línea de código de abajo:  
    // node.next.data = value;
```

```
// Por lo tanto, podemos utilizar el operador de aserción no nulo, !,  
// para afirmar que node.next no está indefinido y silenciar la advertencia del  
compilador  
nodo.siguiente!.datos= valor;  
}
```

# Capítulo 12: Protecciones de tipo definidas por el usuario

## Sección 12.1: Funciones de protección de tipos

Puede declarar funciones que sirvan como guardias de tipo utilizando cualquier lógica que

desees. Toman la forma:

```
function functionName(variableName: any): variableName is DesiredType {  
    // cuerpo que devuelve boolean  
}
```

Si la función devuelve true, TypeScript reducirá el tipo a `DesiredType` en cualquier bloque protegido por una llamada a la función.

Por ejemplo ([pruébalo](#)):

```
function isString(test: any): test es cadena {  
    return typeof prueba === "cadena";  
}  
  
function ejemplo(foo:  
    cualquiera) {  
    if (isString(foo)) {  
        // foo se escribe como cadena en este bloque  
        console.log("es una cadena: " + foo);  
    } else {  
        // foo es el tipo cualquiera en este bloque  
        console.log("¡no sé qué es esto! [" + foo + "]");  
    }  
}  
  
ejemplo("hola mundo"); // imprime "es una cadena: hola  
example({ algo: "else" }); // imprime "¡no sé qué es esto! [[objeto Objeto]]"
```

El predicado de tipo de una función de guarda (el `foo es Bar` en la posición del tipo de retorno de la función) se utiliza en tiempo de compilación para acotar los tipos, el cuerpo de la función se utiliza en tiempo de ejecución. El predicado de tipo y la función deben coincidir, o tu código no funcionará.

Las funciones de protección de tipo no tienen que usar `typeof` o `instanceof`, pueden usar una lógica más

complicada. Por ejemplo, este código determina si tienes un objeto jQuery comprobando su cadena de versión.

```
function isjQuery(foo): foo es JQuery {  
    // prueba de la cadena de versión de  
    // JQuery  
    return foo.jquery !== undefined;  
}  
  
function ejemplo(foo) {  
    if (isjQuery(foo)) {  
        // foo se escribe JQuery aquí  
        foo.eq(0);  
    }  
}
```

## Apartado 12.2: Uso de instanceof

**instanceof** requiere que la variable sea de tipo any. Este

código ([pruébalo](#)):

```
class Mascota { }
class Perro extends Mascota {
  bark() {
    console.log("guau");
  }
}
class Gato extends Mascota {
  purr() {
    console.log("miau");
  }
}

function ejemplo(foo: cualquiera) {
  if (foo instanceof Dog) {
    // foo es el tipo Dog en este bloque
    foo.bark();
  }

  if (foo instanceof Cat) {
    // foo es el tipo Cat en este bloque
    foo.purr();
  }
}

ejemplo( nuevo
Perro());
ejemplo( nuevo Gato());
```

imprime

```
guau
miau
```

a la consola.

## Sección 12.3: Uso de typeof

**typeof** se utiliza cuando es necesario distinguir entre los tipos number, string, boolean y symbol. Otras constantes de cadena no darán error, pero tampoco se utilizarán para restringir tipos.

A diferencia de **instanceof**, **typeof** funcionará con una variable de cualquier tipo. En el ejemplo siguiente, foo podría ser de tipo

number

| sin .

Este código ([pruébalo](#)):

```
function ejemplo(foo: cualquiera) {
  if ( typeof foo === "number" ) {
    // foo es el número de tipo en este bloque
    console.log(foo+ 100);
  }

  if ( typeof foo === "string" ) {
```



```
// foo es de tipo cadena en este bloque  
console.log("no es un número: " + foo);  
}  
}
```

```
ejemplo(23);  
ejemplo("foo");
```

imprime

```
123  
no es un número: foo
```

# Capítulo 13: Ejemplos básicos de TypeScript

## Sección 13.1: 1 ejemplo básico de herencia de clases utilizando las palabras clave extends y super

Una clase genérica Car tiene alguna propiedad de coche y un método de descripción

```
class Car{
    nombre:cadena;
    cilindrada:cadena;

    constructor(nombre:cadena,cilindrada:cadena){
        this.name= name;
        this.engineCapacity= engineCapacity;
    }

    describeCoche(){
        console.log(` $ {this.name} coche viene con$ {this.engineCapacity} desplazamiento`);
    }
}

new Coche("maruti ciaz", "1500cc").describeCoche();
```

HondaCar amplía la clase genérica de coches existente y añade una nueva propiedad.

```
class HondaCar extends Coche{
    Capacidad:number;

    constructor(nombre:cadena,cilindrada:cadena,capacidad:número){ super(nombre,cilindrada);
        this.aforoCapacidad= aforoCapacidad;
    }

    describeHondaCoche(){
        super.describeCoche();
        console.log(` este coche tiene una capacidad de$ {this.seatingCapacity}` );
    }
}

new HondaCar("honda jazz", "1200cc",4).describeHondaCar();
```

## Sección 13.2: 2 ejemplo de variable de clase estática - cuenta cuántas veces se invoca un método

aquí countInstance es una variable estática de clase

```
class StaticTest{
    static countInstance : número= 0;
    constructor(){
        StaticTest.countInstance++;
    }
}

nueva StaticTest();
new StaticTest();
console.log(StaticTest.countInstance);
```

# Capítulo 14: Importación de bibliotecas externas

## Sección 14.1: Búsqueda de ficheros de definición

para typescript 2.x:

las definiciones de [DefinitelyTyped](#) están disponibles a través del paquete [@types npm](#)

```
npm i --save lodash
npm i --save-dev @types/lodash
```

pero en caso de que desee utilizar tipos de otros repos entonces se puede utilizar de forma antigua:

para typescript 1.x:

[Typings](#) es un paquete npm que puede instalar automáticamente archivos de definición de tipos en un proyecto local. Te recomiendo que leas el [quickstart](#).

```
npm install -global typings
```

Ahora tenemos acceso al cli de tipos.

1. El primer paso es buscar el paquete utilizado por el proyecto

tipos b úsqueda lodash				
NOMBRE	FUENTE	PÁGINA DE	DESCRIPCIÓN	VERSIONES
ACTUALI	INICIO			
ZADO	dt	http://lodash.com/		2
2016-07-20T00:13:09.000Z				
lodash	global			1
2016-07-01T20:51:07.000Z				
lodash	npm	https://www.npmjs.com/package/lodash		1
2016-07-01T20:51:07.000Z				

2. A continuación, decida qué fuente debe instalar desde. Yo utilizo dt, que significa [DefinitelyTyped](#), un repositorio de GitHub donde la comunidad puede editar tipografías.
3. Instalar los archivos tipográficos

```
typings install dt~lodash --global --save
```

Vamos a desglosar el último comando. Estamos instalando la versión DefinitelyTyped de lodash como un archivo typings global en nuestro proyecto y guardándolo como una dependencia en `typings.json`. Ahora siempre que importemos `lodash` typescript cargará el archivo de tipado de lodash.

4. Si queremos instalar tipos que sólo se utilizarán en el entorno de desarrollo, podemos utilizar la opción `--save-dev` :

```
typings install chai --save-dev
```

## Sección 14.2: Importar un módulo desde npm

Si dispone de un archivo de definición de tipos (d.ts) para el módulo, puede utilizar una sentencia `import`.

```
import _ = require('lodash');
```

Si no tienes un archivo de definición para el módulo, TypeScript lanzará un error en la compilación porque no puede encontrar el módulo que estás intentando importar.

En este caso, puede importar el módulo con la función `require` normal en tiempo de ejecución. Sin embargo, esto lo devuelve como el tipo `any`.

```
// La variable _ es de tipo any, por lo que TypeScript no realizará ninguna comprobación de tipo.
const _: any = require('lodash');
```

A partir de TypeScript 2.0, también puedes usar una *declaración de módulo ambiental abreviada* para decirle a TypeScript que un módulo existe cuando no tienes un archivo de definición de tipos para el módulo. Sin embargo, TypeScript no será capaz de proporcionar ninguna comprobación tipográfica significativa en este caso.

```
declarar módulo "lodash";

// ahora puedes importar desde lodash de la forma que desees:
import { flatten } from "lodash";
import * as _ from "lodash";
```

A partir de TypeScript 2.1, las reglas se han relajado aún más. Ahora, siempre que un módulo exista en tu directorio `node_modules`, TypeScript te permitirá importarlo, incluso sin declaración de módulo en ninguna parte. (Ten en cuenta que si usas la opción de compilador `--noImplicitAny`, lo siguiente aún generará una advertencia).

```
// Funcionará si `node_modules/someModule/index.js` existe, o si
`node_modules/someModule/package.json` tiene un punto de entrada "main" válido
import { foo } from "algunModulo";
```

## Sección 14.3: Uso de bibliotecas externas globales sin tipado

Aunque lo ideal son los módulos, si la biblioteca que estás utilizando está referenciada por una variable global (como `$` o `_`), porque fue cargada por una etiqueta `script`, puedes crear una declaración de entorno para referirte a ella:

```
declare const _: any;
```

## Sección 14.4: Encontrar archivos de definición con TypeScript 2.x

Con las versiones 2.x de TypeScript, las tipificaciones están ahora disponibles en [el repositorio @types de npm](#). Estas son resueltas automáticamente por el compilador de TypeScript y son mucho más sencillas de usar.

Para instalar una definición de tipo, basta con instalarla como dependencia de desarrollo en el `package.json` del proyecto.

Por ejemplo

```
npm i -S lodash
npm i -D @types/lodash
```

después de la instalación sólo tiene que utilizar el módulo como antes

```
import * as _ from 'lodash'
```

# Capítulo 15: Módulos: exportación e importación

## Sección 15.1: Módulo Hola mundo

```
//hola.ts
export function hola(nombre: cadena){
    console.log(` ¡Hola ${nombre}!` );
}
function holaES(nombre: cadena){
    console.log(` ¡Hola$ {nombre}!` );
}
export {helloES};
export default hola;
```

Carga mediante índice de directorio

Si el directorio contiene un archivo llamado `index.ts` puede ser cargado usando sólo el nombre del directorio (para `index.ts` el nombre del archivo es opcional).

```
//bienvenida/index.ts
export function welcome(name: string){
    console.log(` ¡Bienvenido ${nombre}!` );
}
```

Ejemplo de uso de módulos definidos

```
import {hello, helloES} from "./hello";           // cargar los elementos especificados
import defaultHello from "./hola";                 // carga la exportación por defecto en el nombre
defaultHello                                       // cargar todas las exportaciones como Bundle
import * as Bundle from "./hola";                 // nota index.ts se omite
import {welcome} from "./bienvenida";

hello("Mundo");                                    // ¡Hola Mundo!
helloES("Mundo");                                  // ¡Hola Mundo!
defaultHello("Mundo");                             // ¡Hola Mundo!

Bundle.hello("Mundo");                             // ¡Hola Mundo!
Bundle.helloES("Mundo");                           // ¡Hola Mundo!

welcome("Humano");                                 // ¡Bienvenido
                                                Humano!
```

## Sección 15.2: Reexportación

TypeScript permite reexportar declaraciones.

```
//Operator.ts
interfaz Operador {
    eval(a: número, b: número): número;
}
exportar Operador por defecto;

//Add.ts
import Operador from "./Operator"; export
class A ñadir implements Operador {
    eval(a: número, b: número): número {
        devolver a+ b;
    }
}
```

```

}

//Mul.ts
import Operator from "./Operator"; export
class Mul implements Operator {
    eval(a: n úmero, b: n úmero): n úmero {
        devuelve a * b;
    }
}

```

Puede agrupar todas las operaciones en una sola biblioteca

```

//Operators.ts
import {Add} from "./Add";
import {Mul} from "./Mul";

export {Add, Mul};

```

Las declaraciones con nombre pueden reexportarse utilizando una sintaxis más corta

```

//NamedOperators.ts
export {Add} from "./Add";
export {Mul} from "./Mul";

```

Las exportaciones por defecto también se pueden exportar, pero no se dispone de una sintaxis abreviada. Recuerda que sólo es posible una exportación por defecto por módulo.

```

//Calculator.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
importa Operator de "./Operator";

exportar Operador por defecto;

```

Es posible reexportar la importación agrupada

```

//RepackedCalculator.ts
exportar * de "./Operadores";

```

Al reexportar bundle, las declaraciones pueden anularse cuando se declaran explícitamente.

```

//CalculadoraFija.ts
export * from "./Calculadora"
import Operator from "./Calculadora";
export class Añadir implements Operator {
    eval(a: n úmero, b: n úmero): n úmero {
        devolver 42;
    }
}

```

Ejemplo de uso

```

//run.ts
import {Add, Mul} from "./CalculadoraFija";

const add= new Add();
const mul= new Mul();

```

```
console.log(add.eval(1, 1)); // 42
console.log(mul.eval(3, 4)); // 12
```

## Apartado 15.3: Declaraciones de exportación/importación

Cualquier declaración (variable, const, función, clase, etc.) puede ser exportada desde un módulo para ser importada en otro módulo.

TypeScript ofrece dos tipos de exportación: con nombre y por defecto.

Exportación con nombre

```
// adams.ts
export function hola(nombre: cadena){
    console.log(` ¡Hola ${nombre}!` );
}
export const answerToLifeTheUniverseAndEverything= 42;
export const unused= 0;
```

Al importar exportaciones con nombre, puede especificar qué elementos desea importar.

```
import {hello, answerToLifeTheUniverseAndEverything} from "./adams";
hello(answerToLifeTheUniverseAndEverything); // ¡Hola 42!
```

Exportación por defecto

Cada módulo puede tener una exportación por defecto

```
// dent.ts
const defaultValue= 54;
export default defaultValue;
```

que puede importarse mediante

```
import dentValue from "./dent";
console.log(dentValue); // 54
```

Importación de paquetes

TypeScript ofrece un método para importar un módulo completo en una variable

```
// adams.ts
export function hola(nombre: cadena){
    console.log(` ¡Hola ${nombre}!` );
}
export const answerToLifeTheUniverseAndEverything= 42;

import * as Bundle from "./adams";
Bundle.hello(Bundle.answerToLifeTheUniverseAndEverything); // ¡Hola 42!
console.log(Bundle.unused); // 0
```



# Capítulo 16: Publicar archivos de definición de TypeScript

## Sección 16.1: Incluir archivo de definición con biblioteca en npm

Añada tipos a su package.json

```
{  
  ...  
  "typings": "ruta/archivo.d.ts"  
  ...  
}
```

Ahora cada vez que esa librería sea importada typescript cargará el archivo typings

# Capítulo 17: Uso de TypeScript con webpack

## Sección 17.1: webpack.config.js

install loaders npm **install --save-dev** ts-loader source-map-loader

tsconfig.json

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react"    // si quieres usar react jsx
  }
}

module.exports = {
  entrada: "./src/index.ts",
  salida: {
    nombre de archivo: "./dist/bundle.js",
  },

  // Habilitar mapas de fuentes para depurar la salida de webpack.
  devtool: "source-map",

  resolver: {
    // Añadir '.ts' y '.tsx' como extensiones resolubles.
    extensiones: [".", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
  },

  módulo: {
    cargadores: [
      // Todos los archivos con extensión '.ts' o '.tsx' serán manejados por 'ts-loader'.
      {test: /\.tsx?$/, cargador: "ts-loader"}
    ],

    preLoaders: [
      // Todos los archivos '.js' de salida tendrán cualquier mapa fuente reprocesado por 'source-map-loader'.
      {test: /\.js$/, loader: "source-map-loader"}
    ]
  },

  /*****
   * Si quieres usar react *
   *****/

  // Al importar un módulo cuya ruta coincida con una de las siguientes, simplemente
  // asume que existe una variable global correspondiente y la utiliza en su lugar.
  // Esto es importante porque nos permite evitar la agrupación de todos nuestros
  // dependencias, lo que permite a los navegadores almacenar en caché esas bibliotecas entre compilaciones.
  // externos: {
  //   "react": "React",
  //   "react-dom": "ReactDOM"
  // },
};
```

# Capítulo 18: Mixins

Parámetro	Descripción
derivedCtor	La clase que desea utilizar como clase de composición
baseCtors	Un array de clases a añadir a la clase de composición

## Sección 18.1: Ejemplo de Mixins

Para crear mixins, basta con declarar clases ligeras que puedan utilizarse como "comportamientos".

```
class Moscas {
  fly() {
    alert('¿Es un pájaro? ¿Es un avión?');
  }
}

class Subidas {
  subida() {
    alert('Me hormiguea el sentido arácnido');
  }
}

class Antibalas {
  deflect() {
    alert('Mis alas son un escudo de acero');
  }
}
```

A continuación, puede aplicar estos comportamientos a una clase de composición:

```
class BeetleGuy implements Subidas, Antibalas {
  subida: () => void;
  deflect: () => void;
}

applyMixins(BeetleGuy, [Trepas, Antibalas]);
```

La función `applyMixins` es necesaria para realizar el trabajo de composición.

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach(baseCtor => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach(nombre => {
      if (nombre !== 'constructor') {
        derivedCtor.prototype[nombre] = baseCtor.prototype[nombre];
      }
    });
  });
}
```

# Capítulo 19: Cómo utilizar una biblioteca JavaScript sin un archivo de definición de tipos

Aunque algunas bibliotecas JavaScript existentes tienen [archivos de definición de tipos](#), hay muchas que no los tienen. TypeScript ofrece un par de patrones para manejar las declaraciones que faltan.

## Sección 19.1: Crear un módulo que exporte un valor por defecto any

Para proyectos más complicados, o en casos en los que se pretenda escribir gradualmente una dependencia, puede ser más crear un módulo.

Utilizando como ejemplo JQuery ([aunque dispone de tipados](#)):

```
// place in jquery.d.ts
declare let $ : any;
export default $;
```

Y luego en cualquier archivo en su proyecto, puede importar esta definición con:

```
// algún otro archivo .ts
import $ from "jquery";
```

Después de esta importación, \$ se escribirá como cualquiera .

Si la biblioteca tiene varias variables de nivel superior exporte e importe por nombre:

```
// colocar en jquery.d.ts
declare module "jquery" {
    let $ : any;
    dejar jQuery: any;

    export { $ };
    export { jQuery };
}
```

A continuación, puedes importar y utilizar ambos nombres:

```
// algún otro archivo .ts
import { $ , jQuery } de "jquery";

$.doThing();
jQuery.doOtherThing();
```

## Sección 19.2: Declarar un any global

A veces es más fácil simplemente declarar un global de tipo any, especialmente en proyectos simples. Si jQuery no tuviera declaraciones de tipo ([las tiene](#)), podrías poner

```
declare var $ : any;
```

Ahora cualquier uso de \$ se escribirá any.

## Sección 19.3: Utilizar un módulo ambiental

Si sólo desea indicar la *intención* de una importación (por lo que no desea declarar un global) pero no desea molestarse con ninguna definición explícita, puede importar un módulo de entorno.

```
// en un archivo de declaraciones (como declarations.d.ts)  
declarar m ódulo "jquery"; // observa que no hay exportaciones definidas
```

A continuación, puede importar desde el módulo ambiente.

```
// algún otro archivo .ts  
import {$ , jQuery} de "jquery";
```

Cualquier cosa importada del módulo declarado (como\$ y jQuery ) anteriormente será de tipo any

# Capítulo 20: TypeScript instalación de typescript y ejecución del compilador de typescript tsc

Cómo instalar TypeScript y ejecutar el compilador TypeScript contra un archivo .ts desde la línea de comandos.

## Sección 20.1: Pasos

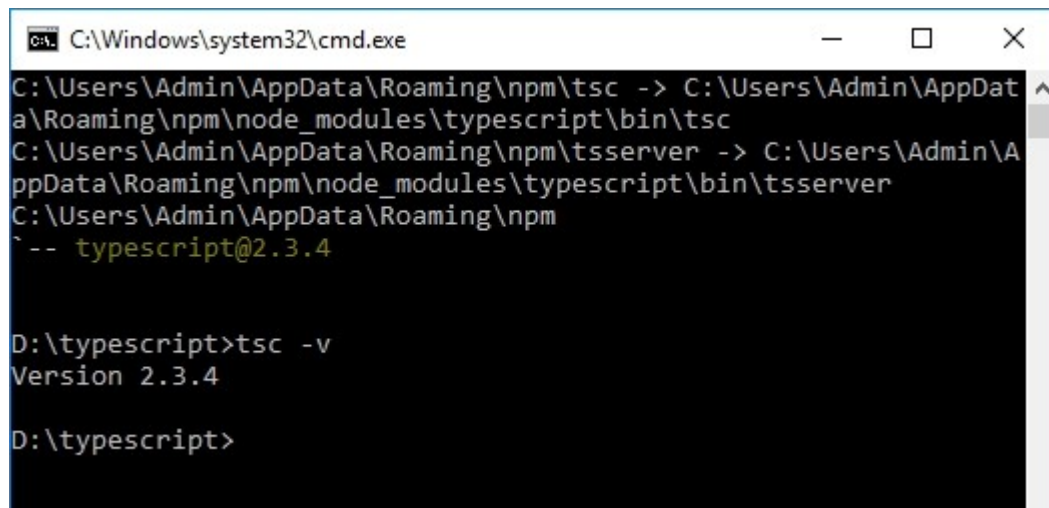
Instalar TypeScript y ejecutar el compilador de TypeScript.

Para instalar el compilador de TypeScript

```
npm install -g typescript
```

Para comprobar con la versión typescript

```
tsc -v
```



```
C:\Windows\system32\cmd.exe
C:\Users\Admin\AppData\Roaming\npm\tsc -> C:\Users\Admin\AppData\Roaming\npm\node_modules\typescript\bin\tsc
C:\Users\Admin\AppData\Roaming\npm\tsserver -> C:\Users\Admin\AppData\Roaming\npm\node_modules\typescript\bin\tsserver
C:\Users\Admin\AppData\Roaming\npm
^-- typescript@2.3.4

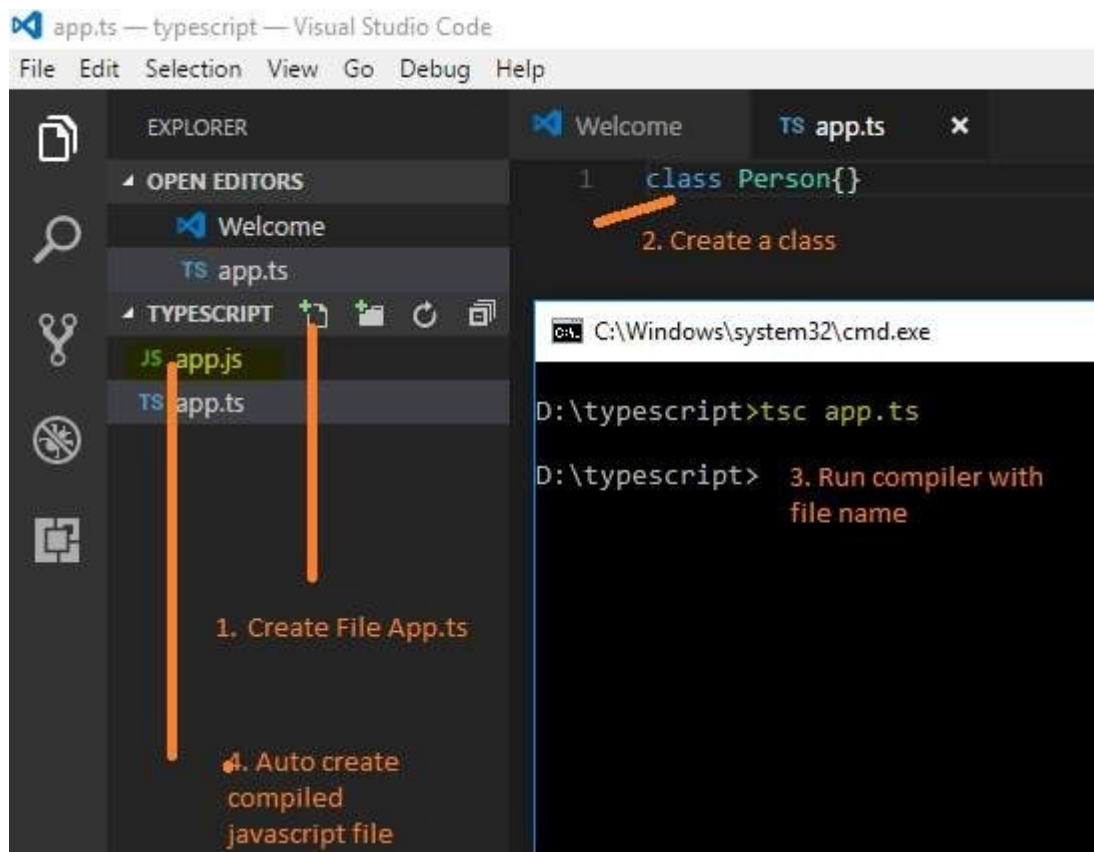
D:\typescript>tsc -v
Version 2.3.4

D:\typescript>
```

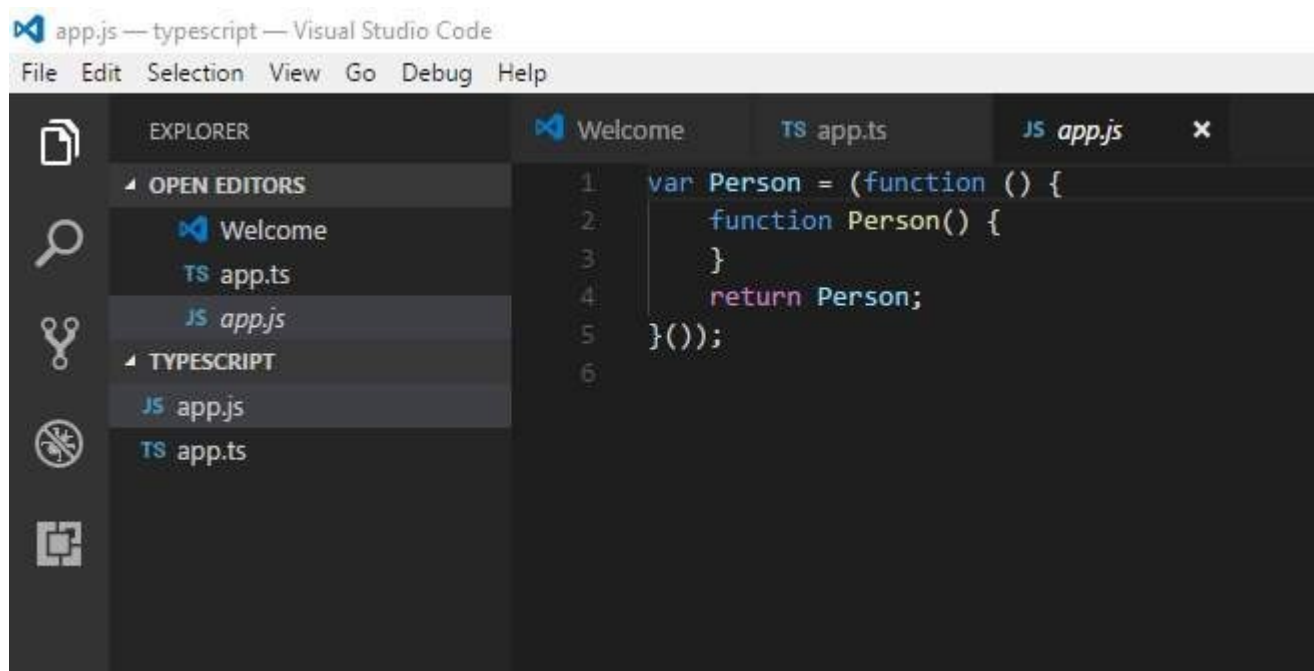
Descargar Visual Studio Code para Linux/Windows

[Enlace de descarga de Visual Code](#)

1. Abrir código de Visual Studio
2. Abra la misma carpeta donde ha instalado el compilador TypeScript
3. Añadir archivo haciendo clic en el icono más del panel izquierdo
4. Crear una clase básica.
5. Compile su archivo de script de tipo y genere la salida.



Vea el resultado en javascript compilado del código typescript escrito.



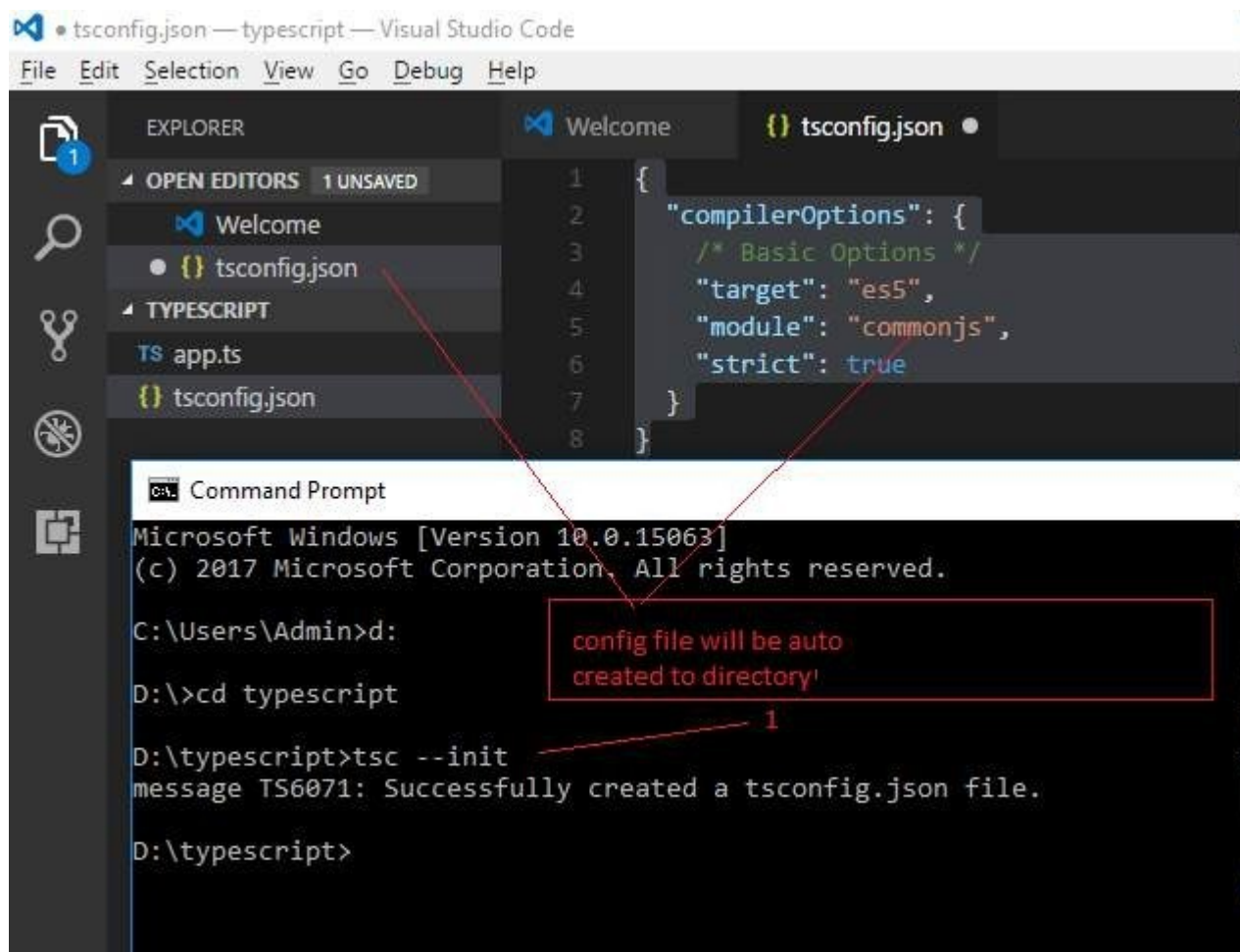
Gracias, señor.

# Capítulo 21: Configurar el proyecto typescript para compilar todos los archivos en typescript.

crear su primer archivo de configuración .tsconfig que le dirá al compilador TypeScript cómo tratar sus archivos .ts

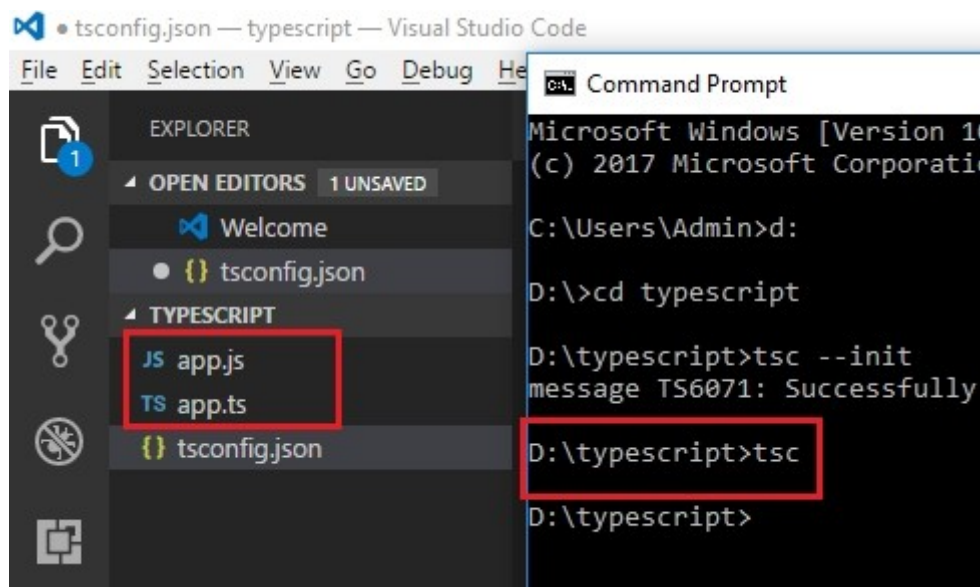
## Sección 21.1: Configuración del archivo de configuración de TypeScript

- Introduce el comando "tsc --init " y pulsa enter.
- Antes de eso tenemos que compilar el archivo ts con el comando "tsc app.ts " ahora todo está definido en el archivo de configuración de abajo de forma automática.



- Ahora, puede compilar todos los typescripts con el comando "tsc ". automáticamente se creará el archivo ".js" de su archivo typescript.





- Si crea otro archivo typescript y pulsa el comando "tsc" en el símbolo del sistema o en el terminal, se creará automáticamente un archivo javascript para el archivo typescript.

Gracias, señor,

# Capítulo 22: Integración con herramientas de compilación

## Sección 22.1: Browserify

Instale

```
npm install tsify
```

Uso de la interfaz de línea de comandos

```
browserify main.ts -p [ tsify --noImplicitAny ]> bundle.js
```

Uso de la API

```
var browserify= require("browserify");
var tsify= require("tsify");

browserify()
  .add("main.ts")
  .plugin("tsify", { noImplicitAny: true })
  .bundle()
  .pipe(proceso.stdout);
```

Más información: [smrgq/tsify](https://github.com/smrgq/tsify)

## Sección 22.2: Webpack

Instale

```
npm install ts-loader --save-dev
```

Webpack .config.js básico

webpack 2.x, 3.x

```
module.exports= {
  resolver: {
    extensiones: ['.ts', '.tsx', '.js']
  },
  módulo: {
    reglas: [
      {
        // Configurar ts-loader para archivos .ts/.tsx y excluir cualquier importación de
        // node_modules.
        prueba: /\.tsx?$/,
        loaders: ['ts-loader'],
        exclude: /node_modules/
      }
    ]
  },
  entrada: [
    // Establecer index.tsx como punto de entrada de la aplicación.
    './index.tsx'
  ],
  salida: {
    nombre de archivo: "bundle.js"
  }
};
```

webpack 1.x

```
module.exports= {
  entrada: './src/index.tsx',
  salida: {
    nombre de archivo: "bundle.js"
  }
};
```

```

},
resolver: {
  // Añadir '.ts' y '.tsx' como extensión resoluble.
  extensiones: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
},
módulo: {
  cargadores: [
    // todos los archivos con extensión '.ts' o '.tsx' serán manejados por 'ts-loader'
    { prueba: /\.ts(x)?$/, loader: "ts-loader", exclude: /node_modules/ }
  ]
}
}

```

Ver más detalles sobre [ts-loader aquí](#).

Alternativas:

- [awesome-typescript-loader](#)

## Sección 22.3: Grunt

Instale

```
npm install grunt-ts
```

Gruntfile.js básico

```

module.exports = function(grunt) {
  grunt.initConfig({
    ts: {
      por defecto : {
        src: ["**/*.ts", "!node_modules/**/*.ts"]
      }
    }
  });
  grunt.loadNpmTasks("grunt-ts");
  grunt.registerTask("default", ["ts"]);
};

```

Más detalles: [TypeStrong/grunt-ts](#)

## Sección 22.4: Gulp

Instale

```
npm install gulp-typescript
```

Gulpfile.js básico

```

var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
  var tsResult = gulp.src("src/*.ts")
    .pipe(ts({
      noImplicitAny: true,
      out: "output.js"
    }));
  return tsResult.js.pipe(gulp.dest("built/local"));
});

```

gulpfile.js usando un tsconfig.json existente

```
var gulp = require("gulp");
```

```

var ts= require("gulp-typescript");

var tsProject= ts.createProject('tsconfig.json', {
  noImplicitAny:    true // Aquí puedes añadir y sobrescribir parámetros
});

gulp.task("default",    function () {
  var tsResult= tsProject.src()
    .pipe(tsProject());
  return tsResult.js.pipe(gulp.dest('release'));
});

```

Más información: [ivogabe/gulp-typescript](http://ivogabe.github.io/gulp-typescript)

## Sección 22.5: MSBuild

Actualizar el archivo de proyecto para incluir Microsoft.TypeScript.Default.props ☐ instalado localmente (en la parte superior) y Microsoft.TypeScript. ☐ (en la parte inferior):

```

<?xml version= "1.0" encoding= "utf-8"    ?>
<Proyecto ToolsVersion= "4.0" DefaultTargets= "Build" xmlns=
"http://schemas.microsoft.com/developer/msbuild/2003"    >
  <!-- Incluir accesorios por defecto en la parte inferior -->
  <Importar

    Proyecto= "$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.Default.props"

    Condition= "Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.Default.props')"    />

  <!-- Las configuraciones de TypeScript van aquí -->
  <PropertyGroup Condición= "$(Configuración)'== 'Depurar'" >
    <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
    <TypeScriptSourceMap>true</TypeScriptSourceMap>
  </PropertyGroup>
  <PropertyGroup Condición= "$(Configuración)'== 'Liberar'" >
    <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
    <TypeScriptSourceMap>false</TypeScriptSourceMap>
  </PropertyGroup>

  <!-- Incluir objetivos por defecto en la parte inferior -->
  <Importar

    Proyecto= "$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets"

    Condition= "Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets')"    />
</Proyecto>

```

Más detalles sobre la definición de las opciones del compilador de MSBuild: [Definición de opciones de compilador en proyectos MSBuild](#)

## Sección 22.6: NuGet

- Haga clic con el botón derecho -> Administrar paquetes NuGet
- Buscar Microsoft.TypeScript.MSBuild ☐
- Pulsa Instalar ☐
- Una vez finalizada la instalación, ¡reconstruye!

Encontrará más información en [Diálogo del gestor de paquetes](#) y [uso de compilaciones nocturnas con NuGet](#)

## Sección 22.7: Instalar y configurar los cargadores webpack +

### Instalación

```
npm install -D webpack typescript ts-loader
```

### webpack.config.js

```
module.exports = {
  entrada: {
    app: ['./src/'],
  },
  salida: {
    ruta: dirname,
    nombre de archivo: './dist/[nombre].js',
  },
  resolver: {
    extensiones: ['', '.js', '.ts'],
  },
  módulo: {
    loaders: [{
      prueba: /\.ts(x)$/, loaders: ['ts-loader'], exclude: /node_modules/
    }],
  }
};
```

# Capítulo 23: Uso de TypeScript con RequireJS

RequireJS es un cargador de archivos y módulos JavaScript. Está optimizado para su uso en navegadores, pero puede utilizarse en otros entornos JavaScript, como Rhino y Node. El uso de un cargador de scripts modular como RequireJS mejorará la velocidad y la calidad de su código.

El uso de TypeScript con RequireJS requiere la configuración de tsconfig.json, y la inclusión de un fragmento en cualquier archivo HTML. El compilador traducirá las importaciones de la sintaxis de TypeScript al formato de RequireJS.

## Sección 23.1: Ejemplo HTML usando RequireJS CDN para incluir un archivo TypeScript ya compilado

```
<body onload= " init();" >
  ...
  <script src= " http://requirejs.org/docs/release/2.3.2/comments/require.js"></script>
  <script>
    function init() {
      require(["view/index.js"]);
    }
  </script>
</body>
```

## Sección 23.2: tsconfig.json ejemplo para compilar en la carpeta view usando el estilo de importación RequireJS

```
{
  "módulo": "amd",           // Usando el generador de código de módulos AMD que funciona con
  "rootDir": "./src",       // RequireJS
  "outDir": "./view",       // Cambialo por tu carpeta fuente
  ...
}
```

# Capítulo 24: TypeScript con AngularJS

Nombre	Descripción
controllerAs	es un nombre de alias al que se pueden variables o funciones. @ver: <a href="https://docs.angularjs.org/guide/directive">https://docs.angularjs.org/guide/directive</a>
\$injector	Lista de inyección de dependencias, es resuelta por angular y pasada como argumento a las funciones constructoras.

## Sección 24.1: Directiva

```
interfaz IMyDirectiveController {  
    // especifique aquí los métodos y propiedades expuestos del controlador  
    getUrl(): cadena;  
}  
  
class MiControladorDirectivo implements IMyControladorDirectivo {  
  
    // Inyecciones internas, por cada directiva  
    public static $inject= ["$ubicación", "tostadora"];  
  
    constructor(private $location: ng.ILocationService, private toaster: any) {  
        // $location y toaster son ahora propiedades del controlador  
    }  
  
    public getUrl(): cadena {  
        return this.$location.url();    // utilizar $location para recuperar laURL  
    }  
}  
  
/*  
 * Inyecciones exteriores, para el control de la carrera.  
 * Por ejemplo, tenemos todas las plantillas en un valor y queremos .  
 */  
export function miDirectiva(templatesUrl: ITemplates): ng.IDirective {  
    devolver {  
        controlador: MyDirectiveController,  
        controllerAs: "vm",  
  
        enlace: ( ámbito: ng.IScope,  
                elemento: ng.IAugmentedJQuery,  
                atributos: ng.IAttributes,  
                controlador: IMyDirectiveController): void=> {  
  
            let url= controlador.getUrl();  
            element.text("URL actual: " + url);  
  
        },  
  
        replace: true, require:  
            "ngModel", restrict:  
            "A",  
        templateUrl: templatesUrl.myDirective,  
    };  
}  
  
myDirective.$inject = [  
    Templates.prototype.slug,  
];
```

```
// El uso de la nomenclatura slug en todos los proyectos simplifica el cambio del nombre de la directiva
miDirectiva. prototype.slug= "miDirectiva";

// Puedes colocar esto en algún archivo bootstrap, o tenerlos en el mismo archivo
angular.module("myApp").
  directive(miDirectiva. prototype.slug, myDirective);
```

## Apartado 24.2: Ejemplo sencillo

```
export function miDirectiva($ubicacion: ng.ILocalizacionServicio): ng.IDirectiva {
  devolver {

    enlace: ( ámbito: ng.IScope,
              elemento: ng.IAugmentedJQuery,
              attributes: ng.IAttributes): void => {

      element.text("URL actual: " + $location.url());

    },

    replace: true,
    require: "ngModel",
    restrict: "A",
    templateUrl: templatesUrl.myDirective,
  };
}
```

```
// El uso de la nomenclatura slug en todos los proyectos simplifica el cambio del nombre dela directiva
miDirectiva. prototype.slug= "miDirectiva";

// Puedes colocar esto en algún archivo bootstrap, o tenerlos en el mismo archivo
angular.module("miApp").
  directive(miDirectiva. prototype.slug, [
    Plantillas. prototype.slug,
    miDirectiva
  ]);
```

## Sección 24.3: Componente

Para una transición más fácil a Angular 2, se recomienda utilizar Component disponible desde Angular 1.5.8

miModulo.ts

```
import { MyModuleComponent } from "../components/myModuleComponent"; import
{ MyModuleService } from "../services/MyModuleService";

angular
  .module("miModulo", [])
  .component("myModuleComponent", new MyModuleComponent())
  .service("miServicioModulo", MiServicioModulo);
```

components/miModuloComponente.ts

```
import IComponentOptions= angular.IComponentOptions;
import IControllerConstructor= angular.IControllerConstructor;
import Injectable= angular.Injectable;
import { MyModuleController } from "../controller/MyModuleController";

export class MiModuloComponente implements IComponenteOpciones {
```



```

    public templateUrl: string = "../app/myModule/templates/myComponentTemplate.html";
    public controller: Injectable< IControllerConstructor> = MyModuleController;
    bindings público: {[boundProperty: cadena]: cadena} = {};
}

```

templates/miComponenteModulo.html

```

<div clase= "mi-m ódulo-componente" >
    {{$ctrl.someContent}}
</div>

```

controlador/MyModuleController.ts

```

import IController= angular.IController;
import { MyModuleService } from "../services/MyModuleService";

export class MiModuloControlador implements IControlador {
    public static readonly $inject: string[] = ["$elemento", "miServicioModulo"]; public
    algunContenido: string= "Hola Mundo";

    constructor($elemento: JQuery, private miServicioModulo: MiServicioModulo) {
        console.log("elemento", $elemento);
    }

    public hacerAlgo(): void {
        // implementación..
    }
}

```

services/MyModuleService.ts

```

export class MiServicioModulo {
    public static readonly $inject: cadena[] = [];

    constructor() {
    }

    public hacerAlgo(): void {
        // hacer algo
    }
}

```

en algún lugar.html

```

<mi-módulo-componente></mi-módulo-componente>

```

# Capítulo 25: TypeScript con SystemJS

## Sección 25.1: Hello World en el navegador con SystemJS

Instalar systemjs y plugin-typescript

```
npm install systemjs
npm install plugin-typescript
```

NOTA: esto instalará el compilador typescript 2.0.0 que aún no ha sido liberado. Para TypeScript 1.8 tienes que usar plugin-typescript 4.0.16

Crear archivo **hello.ts**

```
exportar función greeter(persona: String) {
  return 'Hola, ' + persona;
}
```

Crear el archivo **hello.html**

```
<!doctype html >
<html>
<head>
  <title>Hola Mundo en TypeScript</title>
  <script src= "node_modules/systemjs/dist/system.src.js" "></script>

  <script src= "config.js" "></script>

  <script>
    window.addEventListener('load', function() {
      System.import('./hello.ts').then(function(hola) {
        document.body.innerHTML= hello.greeter('Mundo');
      });
    });
  </script>

</head>
<body>
</body>
</html>
```

Crear **config.js** - Archivo de configuración de SystemJS

```
System.config ({
  paquetes: {
    "plugin-typescript": {
      "main": "plugin.js"
    },
    "typescript": {
      "main": "lib/typescript.js",
      "meta": {
        "lib/typescript.js": {
          "exports": "ts"
        }
      }
    }
  }
})
```

```

    },
    mapa: {
      "plugin-typescript": "node_modules/plugin-typescript/lib/",
      /* NOTA: esto es para npm 3 (node 6) */
      /* para npm 2, la ruta typescript ser á */
      /* node_modules/plugin-typescript/node_modules/typescript */
      "typescript": "node_modules/typescript/"
    },
    transpiler: "plugin-typescript", meta:
    {
      "./hello.ts": {
        formato: "esm",
        cargador: "plugin-typescript"
      }
    },
    typescriptOptions: {
      typeCheck: 'strict'
    }
  }
});

```

NOTA: si no desea la comprobación de tipos, elimine loader: "plugin-typescript" y typescriptOptions de config.js. También tenga en cuenta que nunca comprobará el código javascript, en particular el código en la etiqueta <script> en html ejemplo.

Pruébalo

```

npm install live-server
./node_modules/.bin/live-server --open=hello.html

```

Construir para la producción

```

npm install systemjs-builder

```

Crear archivo build.js :

```

var Builder= require('systemjs-builder');
var builder= new Builder();
builder.loadConfig('./config.js').then(function() {
  builder.bundle('./hello.ts', './hello.js', {minify: true});
});

```

construir hello.js a partir de hello.ts

```

node build.js

```

Utilízelo en producción

Simplemente carga hello.js con una etiqueta script antes de usarlo por primera vez

archivo hola-produccion.html :

```

<!doctype html >
<html>
<head>
  <title>Hola Mundo en TypeScript</title>
  <script src= "node_modules/systemjs/dist/system.src.js" "></script>

  <script src= "config.js" "></script>
  <script src= "hola.js" "></script>

```

```
<script>
  window.addEventListener('load', function() {
    System.import('./hello.ts').then(function(hola) {
      document.body.innerHTML= hello.greeter('Mundo');
    });
  });
</script>
```

```
</head>
<body>
</body>
</html>
```

# Capítulo 26: Uso de TypeScript con React (JS y nativo)

## Sección 26.1: Componente ReactJS escrito en TypeScript

Puedes utilizar los componentes de ReactJS fácilmente en TypeScript. Solo tienes que cambiar el nombre de la extensión de archivo 'jsx' a 'tsx':

```
//holaMensaje.tsx:
var HelloMessage= React.createClass({
  render: function() {
    return< div> Hola {this.props.name}</div>;
  }
});

ReactDOM.render(<HelloMessage nombre= "Juan" />, mountNode);
```

Pero para hacer un uso completo de la principal característica de TypeScript (comprobación estática de tipos) debes hacer un par de cosas:

1) convertir React.createClass en una clase ES6:

```
//holaMensaje.tsx:
class HolaMensaje extends React.Component {
  render() {
    return< div> Hola {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage nombre= "Juan" />, mountNode);
```

Para más información sobre la conversión a ES6, [consulte aquí](#)

2) Añadir interfaces Props y State:

```
interfaz Props { name:cadena;
  optionalParam?:n número;
}

interfaz Estado {
  //vacío en nuestro caso
}

class HolaMensaje extends React.Component< Props, State> {
  render() {
    return< div> Hola {this.props.name}</div>;
  }
}

// TypeScript le permitirá crear sin el parámetro opcional
ReactDOM.render(<HelloMessage nombre= "Sebastian" />, mountNode);
// Pero sí comprueba si pasas un parámetro opcional de tipo incorrecto
ReactDOM.render(<HelloMessage name= "Sebastian" optionalParam= 'foo' />, mountNode);
```

Ahora TypeScript mostrará un error si el programador olvida pasar props. O si intenta pasar props que no están definidos en la interfaz.

## Sección 26.2: TypeScript & react & webpack

Instalación global de typescript, typings y webpack

```
npm install -g typescript typings webpack
```

Instalación de cargadores y vinculación de typescript

```
npm install --save-dev ts-loader source-map-loader npm link typescript
```

Enlazar TypeScript permite a ts-loader utilizar su instalación global de TypeScript en lugar de necesitar una copia local separada [typescript doc](#).

instalación de archivos `.d.ts` con typescript 2.x

```
npm i @types/react --save-dev
npm i @types/react-dom --save-dev
```

instalación de archivos `.d.ts` con typescript 1.x

```
typings install --global --save dt~react typings
install --global --save dt~react-dom
```

Archivo de configuración `tsconfig.json`

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react"
  }
}
```

Archivo de configuración `webpack.config.js`

```
module.exports = {
  entrada: "<ruta al punto de entrada>", // por ejemplo ./src/helloMessage.tsx
  salida: {
    filename: "<ruta al archivo bundle>", // por ejemplo ./dist/bundle.js
  },

  // Habilitar mapas de fuentes para depurar la salida de webpack.
  devtool: "source-map",

  resolver: {
    // Añadir '.ts' y '.tsx' como extensiones resolubles.
    extensiones: [".", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"],
  },

  módulo: {
    cargadores: [
      // Todos los archivos con extensión '.ts' o '.tsx' serán manejados por 'ts-loader'.
      {test: /\.tsx?$/, cargador: "ts-loader"}
    ],
  },
}
```

```

preLoaders: [
  // Todos los archivos '.js' de salida tendrán cualquier mapa fuente reprocesado por 'source-
  // map-loader'.
  {test: /\.js$/, loader: "source-map-loader"}
]

// Al importar un módulo cuya ruta coincida con una de las siguientes, simplemente
// asume que existe una variable global correspondiente y la utiliza en su lugar.
// Esto es importante porque nos permite evitar la agrupación de todos nuestros
// dependencias, lo que permite a los navegadores almacenar en caché esas bibliotecas entre
// compilaciones.
externos: {
  "react": "React",
  "react-dom": "ReactDOM"
}; },

```

finalmente ejecuta webpack o webpack -w (para modo watch)

Nota : React y ReactDOM están marcados como externos

# Capítulo 27: TSLint - asegurando la calidad y consistencia del código

TSLint realiza un análisis estático del código y detecta errores y problemas potenciales en el código.

## Sección 27.1: Configuración para reducir los errores de programación

Este ejemplo tslint.json contiene un conjunto de configuraciones para reforzar más tipados, capturar errores comunes o construcciones confusas que son propensas a producir bugs y seguir más las [Coding Guidelines for TypeScript Contributors](#).

Para aplicar estas reglas, incluya tslint en su proceso de compilación y compruebe su código antes de con tsc.

```
{
  "reglas": {
    // Específico de TypeScript
    "member-access": true, // Requiere declaraciones explícitas de visibilidad para los miembros de la
                           // clase.
    "no-any": true, // Desactiva el uso de any como declaración de tipo.
    // Funcionalidad
    "label-position": true, // Sólo permite etiquetas en ubicaciones sensibles.
    "no-bitwise": true, // Desactiva los operadores bitwise.
    "no-eval": true, // Desactiva las invocaciones a la función eval.
    "no-null-keyword": true, // Desactiva el uso de la palabra clave literal null.
    "no-unsafe-finally": true, // Desactiva las sentencias de flujo de control, como return, continue,
    // rompe y lanza finalmente bloques.
    "no-var-keyword": true, // Desactiva el uso de la palabra clave var.
    "radix": true, // Requiere que se especifique el parámetro radix al llamar a parseInt.
    "triple-equals": true, // Requiere === y !== en lugar de == y !=.
    "use-isnan": true, // Impone el uso de la función isNaN() para comprobar referencias NaN en lugar de
    // una comparación con la constante NaN.
    // Estilo
    "class-name": true, // Aplica PascalCased a los nombres de clases e interfaces.
    "interface-name": [ true, "never-prefix" ], // Requiere que los nombres de interfaz empiecen por
    // mayúscula
  },
  "rules": {
    "no-angle-bracket-type-assertion": true, // Requiere el uso de as Type para las aserciones de tipo.
    // en lugar de <Tipo>.
    "no-unnecessary-semicolon": true, // No permite múltiples definiciones de variables en la misma .
    "quotemark": [ true, "double", "avoid-escape" ], // Requiere comillas dobles para literales de cadena.
    "punto y coma": [ true, "always" ], // Impone el uso consistente del punto y coma al final de cada
    // sentencia.
    "variable-name": [ true, "ban-keywords", "check-format", "allow-leading-underscore" ] // Comprueba los
    // nombres de las variables en busca de varios errores. Desactiva el uso de ciertas palabras clave de TypeScript
    // (any, Number, number, String, string, Boolean, boolean, undefined) como variable o parámetro. Sólo permite nombres
    // de variable en mayúsculas o minúsculas. Permite guiones bajos al principio (sólo tiene efecto si "check-
    // formato" especificado).
  }
}
```

## Sección 27.2: Instalación y configuración

Para instalar [tslint](#) ejecute el comando

```
npm install -g tslint
```

Tslint se configura a través del archivo `tslint.json`. Para inicializar la configuración por defecto ejecute el comando



```
tslint --init
```

Para comprobar posibles errores en el archivo ejecute el comando

```
tslint nombrearchivo.ts
```

## Sección 27.3: Conjuntos de reglas TSLint

- [tslint-microsoft-contrib](#)
- [tslint-eslint-rules](#)
- [codelyzer](#)

Yeoman generador soporta todos estos presets y se puede extender también:

- [generator-tslint](#)

## Sección 27.4: Configuración básica de tslint.json

Esta es una configuración básica

`tslint.json` que impide el uso de cualquier

- requiere llaves para las sentencias `if/else/for/do/while`
- requiere el uso de comillas dobles (`"`) para las cadenas

```
{
  "reglas": {
    "no-any": true,
    "rizado": true,
    "quotemark": [ true, "double" ]
  }
}
```

## Apartado 27.5: Utilizar un conjunto de reglas predefinido como predeterminado

`tslint` puede ampliar un conjunto de reglas existente y se suministra con los valores predeterminados `tslint:recommended` y `tslint:latest`.

`tslint:recommended` es un conjunto de reglas estables y con cierta opinión que fomentamos para la programación general en TypeScript. Esta configuración sigue semver, por lo que no tendrá cambios de ruptura a través de versiones menores o parches.

`tslint:latest` extiende `tslint:recommended` y se actualiza continuamente para incluir la configuración de las últimas reglas en cada versión de TSLint. El uso de esta configuración puede introducir cambios de última hora en versiones menores a medida que se activan nuevas reglas que causan fallos de lint en su código. Cuando TSLint alcance una versión mayor, `tslint:recommended` se actualizará para ser idéntica a `tslint:latest`.

[Docs](#) y [código fuente del conjunto de reglas](#).

[predefinidas](#) Para que uno pueda usarlas

simplemente:

```
{
  "extends": "tslint:recommended"
}
```

para tener una configuración inicial sensata.

A continuación, se pueden sobrescribir las reglas de ese preajuste mediante reglas , por ejemplo, para los desarrolladores de nodos tenía sentido establecer no-console

a **false**:

```
{
  "extends": "tslint:recommended",
  "rules": {
    "no-console": false
  }
}
```

# Capítulo 28: tsconfig.json

## Sección 28.1: Crear proyecto TypeScript con tsconfig.json

La presencia de un archivo `tsconfig.json` indica que el directorio actual es la raíz de un TypeScript habilitado. La inicialización de un proyecto TypeScript, o mejor dicho del archivo `tsconfig.json`, puede realizarse mediante el siguiente comando:

```
tsc --init
```

A partir de TypeScript v2.3.0 y superiores esto creará el siguiente `tsconfig.json` por defecto:

```
{
  "compilerOptions": {
    /* Opciones básicas */
    "target": "es5", /* Especificar la versión de destino ECMAScript: 'ES3' (por defecto), ES5, "ES2015", ES2016, "ES2017" o "ESNEXT". */
    "module": "commonjs", /* Especifica la generación de código del módulo: 'commonjs', 'amd', 'system', 'umd' o 'es2015'. */
    // "lib": [], /* Especificar los archivos de biblioteca que se incluirán en la compilación: */
    // "allowJs": true, /* Permitir la compilación de archivos javascript. */
    // "checkJs": true, /* Informar de errores en archivos .js. */
    // "jsx": "preserve", /* Especifica la generación de código JSX: "preserve", "react-native", o "react". */
    // "declaration": true, /* Genera el archivo '.d.ts' correspondiente. */
    // "sourceMap": true, /* Genera el archivo '.map' correspondiente. */
    // "outFile": "./", /* Concatenar y emitir la salida a un único fichero. */
    // "outDir": "./", /* Redirige la estructura de salida al . */
    // "rootDir": "./", /* Especifica el directorio raíz de los archivos de entrada. Utilízelo para controlar la estructura del directorio de salida con --outDir. */
    // "removeComments": true, /* No emitir comentarios a la salida. */
    // "noEmit": true, /* No emitir salidas. */
    // "importHelpers": true, /* Importar emit helpers de 'tslib'. */
    // "downlevelIteration": true, /* Proporcionar soporte completo para iterables en 'for-of', spread, y destructuring cuando se apunta a 'ES5' o 'ES3'. */
    // "isolatedModules": true, /* Transpile cada archivo como un módulo independiente (similar a 'ts.transpileModule'). */

    /* Opciones de comprobación de tipo estricto */
    "strict": true /* Habilitar todas las opciones de comprobación de tipo estricto. */
    // "noImplicitAny": true, /* Genera un error en expresiones y declaraciones con un tipo 'any' implícito. */
    // "strictNullChecks": true, /* Habilitar comprobaciones nulas estrictas. */
    // "noImplicitThis": true, /* Genera un error en las expresiones 'this' con un tipo 'any' implícito. */
    // "alwaysStrict": true, /* Analiza en modo estricto y emite "use strict" para cada fichero fuente. */

    /* Comprobaciones adicionales */
    // "noUnusedLocals": true, /* Informar de errores en locales no utilizados. */
    // "noUnusedParameters": true, /* Informar de errores en parámetros no utilizados. */
    // "noImplicitReturns": true, /* Informar de error cuando no todas las rutas de código en la función devuelven un valor. */
    // "noFallthroughCasesInSwitch": true, /* Informar de errores para casos fallidos en la sentencia switch. */

    /* Opciones de resolución del módulo */
    // "moduleResolution": "node", /* Especificar la estrategia de resolución del módulo: "node" (Node.js) */
  }
}
```

```

o 'classic' (TypeScript pre-1.6). */
// "baseUrl": "./", /* Directorio base para resolver nombres de módulos no
absolutos.
*/
// "camino": {}, /* Una serie de entradas que reasignan las importaciones a la
ubicaciones relativas a la 'baseUrl'. */
// "rootDirs": [], /* Lista de carpetas raíz cuyo contenido
combinado representa la estructura del proyecto en tiempo de ejecución. */
// "typeRoots": [], /* Lista de carpetas de las que incluir definiciones de tipos.
*/
// "tipos": [], /* Ficheros de declaración de tipos a
incluir en la compilación. */
// "allowSyntheticDefaultImports": true, /* Permitir importaciones por defecto de módulos sin
exportación por defecto. Esto no afecta a la emisión de código, sólo a la comprobación de tipos. */

/* Opciones del mapa de origen */
// "sourceRoot": "./", /* Especificar la ubicación donde el depurador debe
localizar los archivos TypeScript en lugar de las ubicaciones de origen. */
// "mapRoot": "./", /* Especificar la ubicación donde el depurador debe
localizar los archivos de mapa en lugar de las ubicaciones generadas. */
// "inlineSourceMap": true, /* Emite un único fichero con los mapas fuente en
lugar de tener un fichero separado. */
// "inlineSources": true, /* Emite la fuente junto con los mapas de fuentes dentro
de un único archivo; requiere que se establezca '--inlineSourceMap' o '--sourceMap'. */

/* Opciones experimentales */
// "experimentalDecorators": true, /* Habilita el soporte experimental para decoradores ES7. */
// "emitDecoratorMetadata": true, /* Habilita el soporte experimental para emitir
metadatos de tipo para decoradores. */
}
}

```

La mayoría, si no todas, las opciones se generan automáticamente con sólo las necesidades básicas sin comentar.

Las versiones antiguas de TypeScript, como por ejemplo v2.0.x e inferiores, generarían un tsconfig.json como este:

```

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

```

## Sección 28.2: Configuración para reducir los errores de programación

Hay muy buenas configuraciones para forzar la tipificación y obtener errores más útiles que no están activadas por defecto.

```

{
  "compilerOptions": {

    "alwaysStrict": true, // Analiza en modo estricto y emite "use strict" para cada fichero fuente.

    // Si tienes un casing incorrecto en los archivos referenciados p.e. el nombre del archivo es Global.ts
    y tienes un ///
    <reference path="global.ts" /> para hacer referencia a este archivo, entonces esto puede causar a errores
    inesperados.
    Visite: http://stackoverflow.com/questions/36628612/typescript-transpiler-casing-issue
    "forceConsistentCasingInFileNames": true, // No permitir referencias aun mismo fichero con
    tipografía inconsistente

    // "allowUnreachableCode": false, // No informar de errores en código inalcanzable. (Por defecto: False)
    // "allowUnusedLabels": false, // No informar de errores en etiquetas no utilizadas. (Por defecto: False)
  }
}

```

```

"noFallthroughCasesInSwitch":    true, // Informar de errores en los casos fallidos en la sentencia
switch.
"noImplicitReturns":             true, // Informar de error cuando no todas las rutas de código de la función
devuelven un valor.

"noUnusedParameters":          true, // Informar de errores en parámetros no utilizados.
"noUnusedLocals":               true, // Informar de errores en locales no utilizados.

"noImplicitAny":                true, // Lanzar error en expresiones y declaraciones con un tipo "any" implícito.
"noImplicitThis":               true, // Lanzar error en estas expresiones con un tipo "any" implícito.

"strictNullChecks":             true, // Los valores null e undefined no están en el dominio de cada tipo y sólo son
asignables a sí mismos y a any.

// Para aplicar estas reglas, añada esta configuración.
"noEmitOnError":                true    // No emitir salidas si se reportó algún error.
}
}

```

¿No es suficiente? Si eres un codificador duro y quieres más, entonces puede que te interese comprobar tus archivos TypeScript con tslint antes de compilarlos con tsc. Compruebe cómo configurar tslint para un código aún más estricto.

## Sección 28.3: compileOnSave

Establecer una propiedad de nivel superior `compileOnSave` indica al IDE que genere todos los ficheros para un `tsconfig.json` dado al guardar.

```

{
  "compileOnSave": true,
  "compilerOptions": {
    ...
  },
  "excluir": [
    ...
  ]
}

```

Esta característica está disponible desde TypeScript 1.8.4 en adelante, pero necesita ser soportada directamente por los IDEs. Actualmente, ejemplos de IDEs soportados son:

- Visual Studio 2015 [con la actualización 3](#)
- JetBrains WebStorm
- Átomo [con atom-typescript](#)

## Sección 28.4: Observaciones

Un fichero `tsconfig.json` puede contener tanto comentarios de línea como de bloque, usando las mismas reglas que ECMAScript.

```

//Comentario principal
{
  "compilerOptions": {
    //esto es un comentario de línea
    "module": "commonjs", //comentario de línea eol
    "target" /*bloque en línea*/ : "es5",
    /* Esto es un comentario de bloque */
  }
}

```

## Sección 28.5: preserveConstEnums

TypeScript soporta enumerables constantes, declarados a través de `const enum`

Esto suele ser sólo azúcar sintáctico, ya que las constantes enum están en línea en JavaScript

compilado. Por ejemplo, el siguiente código

```
const enum Tristate {  
    Verdadero,  
    Falso,  
    Desconocido  
}  
  
var algo = Tristate.True;
```

se compila en

```
var algo = 0;
```

Aunque el rendimiento se beneficia del inlining, puede que prefiera mantener los enums aunque sean constantes (es decir: puede que desee legibilidad en el código de desarrollo), para ello tiene que establecer en `tsconfig.json` la cláusula `preserveConstEnums` en las `compilerOptions` a `true`.

```
{  
  "compilerOptions": {  
    "preserveConstEnums": true,  
    ...  
  },  
  "excluir": [  
    ...  
  ]  
}
```

De esta forma el ejemplo anterior se compilaría como cualquier otro enums, tal y como se muestra en el siguiente snippet.

```
var Tristate; (  
  function  
(Tristate) {  
    Tristate[Tristate["Verdadero"] = 0] = "Verdadero";  
    Tristate[Tristate["Falso"] = 1] = "Falso";  
    Tristate[Tristate["Desconocido"] = 2] = "Desconocido";  
  })(Tristate || (Tristate = {}));  
  
var algo = Tristate.True
```

# Capítulo 29: Depuración

Hay dos formas de ejecutar y depurar TypeScript:

Transpile a JavaScript , ejecute en node y utilice mapeos para enlazar de nuevo a los archivos fuente de

TypeScript o

Ejecute TypeScript directamente con [ts-node](#)

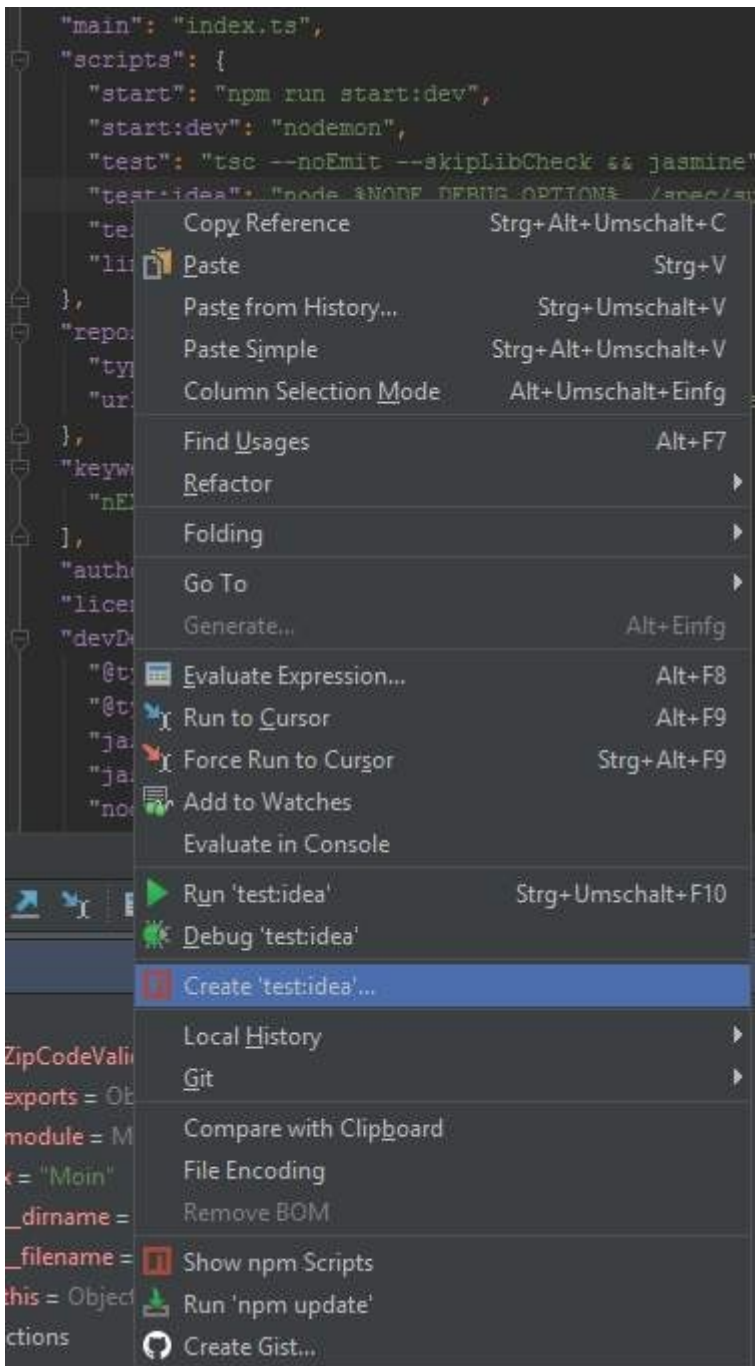
Este artículo describe ambas formas utilizando [Visual Studio Code](#) y [WebStorm](#). Todos los ejemplos suponen que el archivo principal es *index.ts*.

## Sección 29.1: TypeScript con ts-node en WebStorm

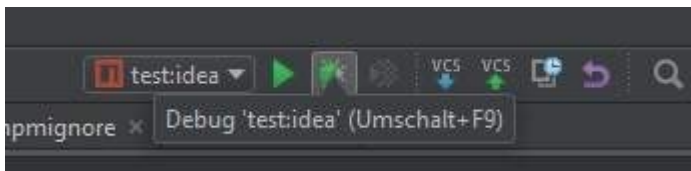
Añade este script a tu `package.json` :

```
"inicio:idea": "ts-node %NODE_DEBUG_OPTION% --ignore false index.ts",
```

Haga clic con el botón derecho en el script y seleccione *Crear 'test:idea'...* y confirme con 'OK' para crear la configuración de depuración:



Inicie el depurador utilizando esta configuración:



## Sección 29.2: TypeScript con ts-node en Visual Studio Code

Añade ts-node a tu proyecto TypeScript:

```
npm i ts-node
```

Añade un script a tu package.json :

```
"start:debug": "ts-node --inspect=5858 --debug-brk --ignore false index.ts"
```



El `launch.json` necesita ser configurado para usar el tipo `node2` e iniciar npm ejecutando el script `start:debug` :

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "tipo": "node2",
      "request": "lanzar",
      "nombre": "Lanzar programa",
      "runtimeExecutable": "npm",
      "windows": {
        "runtimeExecutable": "npm.cmd"
      },
      "runtimeArgs": [
        "run-script",
        "start:debug"
      ],
      "cwd": "${workspaceRoot}/server",
      "outFiles": [],
      "port": 5858, "sourceMaps":
      true
    }
  ]
}
```

## Sección 29.3: JavaScript con SourceMaps en Visual Studio Code

En el `tsconfig.json` establezca

```
"sourceMap": true,
```

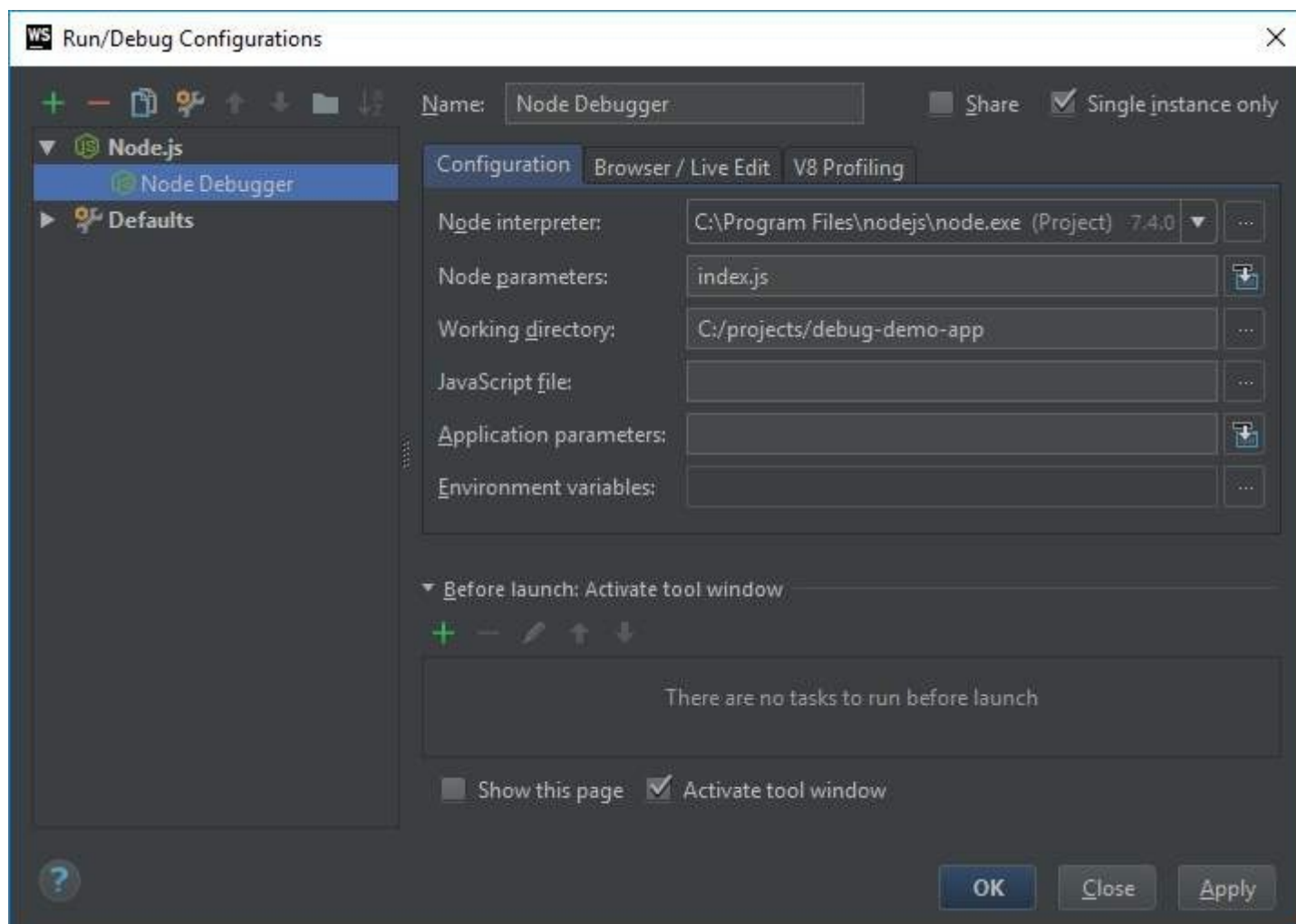
para generar mapeos junto con los archivos js de las fuentes TypeScript utilizando el comando `tsc` . El archivo [launch.json](#):

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "tipo": "nodo",
      "request": "lanzamiento",
      "nombre": "Programa de
      lanzamiento",
      "program": "${workspaceRoot}\\index.js",
      "cwd": "${workspaceRoot}",
      "outFiles": [],
      "sourceMaps": true
    }
  ]
}
```

Esto inicia node con el archivo `index.js` generado (si su archivo principal es `index.ts`) y el depurador en Visual Studio Code que se detiene en los puntos de interrupción y resuelve los valores de las variables dentro de su código TypeScript.

## Sección 29.4: JavaScript con SourceMaps en WebStorm

Crear una [configuración de depuración](#) `Node.js` y utilizar `index.js` como *parámetros Node*.



# Capítulo 30. Pruebas unitarias Pruebas unitarias

## Sección 30.1: cinta

tape es un framework de pruebas JavaScript minimalista, que produce marcado [compatible con](#)

[TAP](#). Para instalar tape usando el comando de ejecución npm

```
npm install --save-dev tape @tipos/tape
```

Para utilizar la cinta con TypeScript es necesario instalar ts-node como paquete global, para ello ejecute el comando

```
npm install -g ts-node
```

Ya está listo para escribir su primera prueba

```
//math.test.ts
import * as test from "cinta";

test("Prueba matemática",
  (t)=> {
    t.equal(4, 2 + 2);
    t.true(5 > 2 + 2);

    t.end();
  });
```

Para ejecutar el comando de ejecución de la prueba

```
ts-node node_modules/tape/bin/tape math.test.ts
```

En la salida debería ver

```
TAP versión 13
# Test de
matemáticas
ok 1 debe ser igual ok
2 debe ser veraz

1..2
# pruebas 2
# pass 2

# ok
```

Buen trabajo, acabas de ejecutar tu prueba TypeScript.

Ejecutar varios archivos de prueba

Puede ejecutar varios archivos de prueba a la vez utilizando comodines de ruta. Para ejecutar todas las pruebas TypeScript en el directorio de pruebas ejecute el comando

```
ts-node node_modules/tape/bin/tape tests/**/*.ts
```

## Sección 30.2: jest (ts-jest)

[jest](#) es un framework de pruebas JavaScript de Facebook, con [ts-jest](#) se puede utilizar para probar código

TypeScript. Para instalar jest usando el comando de ejecución npm

```
npm install --save-dev jest @types/jest ts-jest typescript
```

Para facilitar su uso, instale jest como paquete global

```
npm install -g jest
```

Para que jest funcione con TypeScript es necesario añadir la configuración a package.json

```
//package.json
{
  ...
  "jest": {
    "transformer": {
      "ts|tsx": "<rootDir>/node_modules/ts-jest/preprocessor.js"
    },
    "testRegex": "(/pruebas /.*|\\.(prueba|spec))\\.\\.(ts|tsx|js)$",
    "moduleFileExtensions": ["ts", "tsx", "js"]
  }
}
```

Ahora jest está listo. Supongamos que tenemos el ejemplo fizz buz para probar

```
//fizzBuzz.ts
export function fizzBuzz(n: number): string {
  let output = '';
  for (let i = 1; i <= n; i++) {
    if (i % 5 && i % 3) {
      output += i + ' ';
    }
    if (i % 3 === 0) {
      salida += 'Fizz ';
    }
    if (i % 5 === 0) {
      salida += 'Buzz ';
    }
  }
  return salida;
}
```

Un ejemplo de prueba podría ser

```
//FizzBuzz.test.ts
<reference types="jest" />

import { fizzBuzz } from './fizzBuzz';
test('Prueba FizzBuzz', () => {
  expect(fizzBuzz(2)).toBe('1 2 '); expect(fizzBuzz(3)).toBe('1 2 Fizz ');
});
```

Para ejecutar la prueba

```
jest
```

En la salida debería ver

```
PASS ./fizzBuzz.test.ts
✓ Prueba FizzBuzz (3 ms)

: 1 superada, 1 en total
Pruebas:      1 aprobado, 1 en total
Instantáneas: 0 total
Tiempo:       1.46s, estimado 2s
Ejecuté todas las pruebas.
```

Cobertura del código

jest soporta la generación de informes de cobertura de código.

Para utilizar la cobertura de código con TypeScript es necesario añadir otra línea de configuración a `package.json`.

```
{
  ...
  "jest": {
    ...
    "testResultsProcessor": "<rootDir>/node_modules/ts-jest/coverageprocessor.js"
  }
}
```

Para ejecutar pruebas con generación de informe de cobertura ejecute

```
jest --coverage
```

Si se utiliza con nuestro zumbido fizz muestra debe ver

```
PASS ./fizzBuzz.test.ts
✓ Prueba FizzBuzz (3 ms)

-----|-----|-----|-----|-----|-----|
Fichero  | % Stmts| % Branch| % Funcs| % Lines| Cubrir |
-----|-----|-----|-----|-----|-----|
Todos los archivos  92. |      87. |    100 |    91, |      |
fizzBuzz. |    92. |      87. |    100 |    91, |      |
-----|-----|-----|-----|-----|-----|
: 1 superada, 1 en total
Pruebas:      1 aprobado, 1 en total
Instantáneas: 0 total
Tiempo:       1.857s
Ejecuté todas las pruebas.
```

jest también ha creado una carpeta `coverage` que contiene informes de cobertura en varios formatos, incluido un informe html fácil de usar en `coverage/lcov-report/index.html`

## All files

92.31% Statements 12/13 87.5% Branches 7/8 100% Functions 1/1 91.67% Lines 11/12

File		Statements		Branches		Functions		Lines	
fizzBuzz.ts	<div><div></div></div>	92.31%	12/13	87.5%	7/8	100%	1/1	91.67%	11/12

## Sección 30.3: Alsaciano

[Alsatian](#) es un framework de pruebas unitarias escrito en TypeScript. Permite el uso de casos de prueba y genera un marcado compatible con TAP.

Para utilizarlo, instálelo desde npm

```
npm install alsatian --save-dev
```

A continuación, cree un archivo de prueba:

```
import { Expect, Test, TestCase } de "alsatian"; import
{ SomeModule } de "../src/some-module";

export CiertoModuloPruebas {

  @Pruebas()
  public statusShouldBeTrueByDefault() {
    dejar instancia= nuevo SomeModule();

    Expect(instance.status).toBe( true);
  }

  @Test("El nombre debe ser nulo por defecto")
  public nombreDebeSerNuloPorDefecto() {
    dejar instancia= nuevo SomeModule();

    Expect(instance.name).toBe( null);
  }

  @TestCase("") @TestCase("manzanas")
  public shouldSetNameCorrectly(nombre: cadena) {
    dejar instancia= nuevo SomeModule();

    instance.setName(name);

    Expect(instance.name).toBe(name);
  }
}
```

Para una documentación completa, consulte [el repositorio GitHub de Alsatian](#).

## Sección 30.4: complemento chai-immutable

1. Instalar desde npm chai, chai-immutable, y ts-node

```
npm install --save-dev chai chai-immutable ts-node
```

2. Tipos de instalación para mocha y chai

```
npm install --save-dev @types/mocha @types/chai
```

3. Escriba un archivo de prueba sencillo:

```
import {List, Set} from 'immutable'; import
* as chai from 'chai';
import * as chaiImmutable from 'chai-immutable'; chai.use(chaiImmutable);

describe('chai ejemplo immutable', () => {
  it('ejemplo', () => {
    esperar(Conjunto.de(1,2,3)).que.no.est     é.vac ío;

    esperar(Conjunto.de(1,2,3)).que.incluya(2);
    esperar(Conjunto.de(1,2,3)).que.incluya(5);
  })
})
```

4. Ejecútalo en la consola:

```
mocha --compilers ts:ts-node/register,tsx:ts-node/register 'test/**/*.spec.@(ts|tsx)'
```

# Créditos

Muchas gracias a todas las personas de Stack Overflow Documentation que ayudaron a proporcionar este contenido, se pueden enviar más cambios a [web@petercv.com](mailto:web@petercv.com) para que se publiquen o actualicen nuevos contenidos.

<a href="#">2426021684</a>	Capítulos 1, 14 y 16
<a href="#">ABabin</a>	Capitulo 9
<a href="#">Alec Hansen</a>	Capitulo 1
<a href="#">Alex Filatov</a>	Capítulos 22 y 27
<a href="#">Almendra</a>	Capítulo 14
<a href="#">Aminadav</a>	Capítulo 9
<a href="#">Aron</a>	Capítulo 9
<a href="#">artem</a>	Capítulos 9, 14 y 25
<a href="#">Blackus</a>	Capitulo 14
<a href="#">bnieland</a>	Capítulo 28
<a href="#">br4d</a>	Capítulo 6
<a href="#">BrunoLM</a>	Capítulos 1, 17 y 22
<a href="#">Brutus</a>	Capitulo 14
<a href="#">ChanceM</a>	Capitulo 1
<a href="#">Cobus Kruger</a>	Capitulo 9
<a href="#">danvk</a>	Capítulos 1, 2 y 11
<a href="#">dimitrisli</a>	Capítulo 5
<a href="#">dublicador</a>	Capitulo 14
<a href="#">Equiman</a>	Capítulo 7
<a href="#">Fenton</a>	Capítulos 3 y 18
<a href="#">Florian Hämmerle</a>	Capítulo 5
<a href="#">Fylax</a>	Capítulos 1, 3 y 28
<a href="#">goenning</a>	Capítulo 28
<a href="#">hansmaad</a>	Capítulos 7 y 10
<a href="#">Harry</a>	Capitulo 14
<a href="#">irakli khitarishvili</a>	Capítulos 17 y 26
<a href="#">islandman93</a>	Capítulos 1, 6, 9, 14 y 26
<a href="#">James Monger</a>	Capítulos 7, 27 y 30
<a href="#">JKillian</a>	Capítulos 11 y 14
<a href="#">Joel Day</a>	Capitulo 14
<a href="#">John Ruddell</a>	Capítulo 22
<a href="#">Joshua Breeden</a>	Capítulos 1 y 9
<a href="#">Juliën</a>	Capítulos 3 y 28
<a href="#">Justin Niles</a>	Capítulo 7
<a href="#">k0pernikus</a>	Capítulos 1 y 27
<a href="#">Kevin Montrose</a>	Capítulos 5, 12 y 19
<a href="#">Kewin Dousse</a>	Capítulo 22
<a href="#">KnottytOmo</a>	Capítulos 1, 10 y 14
<a href="#">Kuba Beránek</a>	Capítulo 1
<a href="#">Lekhnaath</a>	Capítulo 1
<a href="#">leonidv</a>	Capítulo 30
<a href="#">lilezek</a>	Capítulo 23
<a href="#">Magu</a>	Capítulos 3, 27 y 28
<a href="#">Matt Lishman</a>	Capitulo 1
<a href="#">Matthew Harwood</a>	Capítulo 30
<a href="#">Mijail</a>	Capítulos 1 y 3
<a href="#">mleko</a>	Capítulos 1, 15, 22, 27 y 30



<a href="#">muetzerich</a>	Capítulo 6
<a href="#">Muhammad Awais</a>	Capítulo 10
<a href="#">Paul Boutes</a>	Capítulo 9
<a href="#">Peopleware</a>	Capítulo 29
<a href="#">Rahul</a>	Capítulos 20 y 21
<a href="#">Rajab Shakirov</a>	Capítulos 14 y 26
<a href="#">RationalDev</a>	Capítulos 1 y 3
<a href="#">Remo H. Jansen</a>	Capítulo 8
<a href="#">Robin</a>	Capitulo 7
<a href="#">Roman M. Koss</a>	Capítulo 24
<a href="#">Roy Dictus</a>	Capitulo 1
<a href="#">Saiful Azad</a>	Capítulos 1 y 9
<a href="#">Sam</a>	Capítulo 1
<a href="#">samAlvin</a>	Capítulo 1
<a href="#">SilentLupin</a>	Capítulo 6
<a href="#">Slava Shpitalny</a>	Capítulos 6, 9, 10 y 14
<a href="#">smnbbrv</a>	Capítulo 5
<a href="#">Stefan Rein</a>	Capítulo 24
<a href="#">Sunnyok</a>	Capítulo 9
<a href="#">Taytay</a>	Capítulo 10
<a href="#">Udlel Nati</a>	Capítulo 4
<a href="#">usuario3893988</a>	Capítulo 28
<a href="#">vashishth</a>	Capítulo 13
<a href="#">Abanico de Wasabi</a>	Capitulo 1

# También le puede interesar

