

TypeScript

Apuntes para Profesionales

Chapter 6: Functions Section 6.1: Optional and Default Parameters

Optional Parameters

In TypeScript, every parameter is assumed to be required by the function. You can add a ? at parameter name to set it as optional.
For example, the lastName parameter of this function is optional:

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

Optional parameters must come after all non-optional parameters:

```
function buildName(firstName: string, lastName: string) // Invalid  
function buildName(firstName?: string, lastName: string) // Invalid
```

Default Parameters

If the user passes undefined or doesn't specify an argument, the default value will be default-initialized parameters.

For example, "Smith" is the default value for the lastName parameter:

```
function buildName(firstName: string, lastName = "Smith") {  
    // ...  
}
```

Section 6.2: Function as a parameter

Suppose we want to receive a function as a parameter, we can do it like this:

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

If we want to receive a constructor as a parameter:

```
function foo(ctor: Function): void {  
    // ...  
}
```

```
function foo(ctor: Function): void {  
    // ...  
}
```

Or to make it easier to read we can define an interface describing the constructor:

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

TypeScript Notes for Professionals

Chapter 7: Classes

TypeScript, like ECMAScript 6, support object-oriented programming using classes. This contrasts with older JavaScript versions, which only supported prototype-based inheritance chain.

The class support in TypeScript is similar to that of languages like Java and C#, in that classes may inherit from other classes, while objects are instantiated as class instances.

Also similar to those languages, TypeScript classes may implement interfaces or make use of generics.

Section 7.1: Abstract Classes

```
abstract class Machine {  
    constructor(public manufacturer: string) {}  
}
```

// An abstract class can define methods of its own. Or...
summary(): string {
 return `This \${this.manufacturer} makes this machine.`;
}

// Require inheriting classes to implement methods
abstract moreInfo(): string;

```
class Car extends Machine {  
    constructor(public manufacturer: string, public position: number, protected speed: number) {  
        super(manufacturer);  
    }  
}
```

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

```
moreInfo() {  
    return `This is a car located at ${this.position} and going ${this.speed}mph!`;  
}
```

```
let myCar = new Car("Honda", 10, 70);  
myCar.moreInfo(); // position is now 80  
console.log(myCar.summary()); // prints "Honda makes this machine."  
console.log(myCar.moreInfo()); // prints "This is a car located at 80 and going 70mph!"
```

Abstract classes are base classes from which other classes can extend. They cannot be instantiated themselves; you cannot do `new Machine("Honda")`.

The two key characteristics of an abstract class in TypeScript are:

1. They can implement methods of their own.
2. They can define methods that inheriting classes **must** implement.

For this reason, abstract classes can conceptually be considered a combination of an interface and a class.

Section 7.2: Simple class

```
class Car {  
    // ...  
}
```

TypeScript Notes for Professionals

Chapter 13: TypeScript basic examples Section 13.1: 1 basic class inheritance example using extends and super keyword

A generic Car class has some car property and a description method

```
class Car {  
    name: string;  
    engineCapacity: string;
```

```
    constructor(name: string, engineCapacity: string) {  
        this.name = name;  
        this.engineCapacity = engineCapacity;
```

```
    describeCar() {  
        console.log(`This is a car with ${this.name} and going with ${this.engineCapacity} displacement`);
```

```
    }  
}
```

```
new Car("Honda", "1000cc").describeCar();
```

HondaCar extends the existing generic car class and adds new property.

```
class HondaCar extends Car {  
    seatingCapacity: number;
```

```
    constructor(name: string, engineCapacity: string, seatingCapacity: number) {  
        super(name, engineCapacity);  
        this.seatingCapacity = seatingCapacity;
```

```
    describeHondaCar() {  
        super.describeCar();  
        console.log(`This car has seating capacity of ${this.seatingCapacity}`);
```

```
    }  
}
```

```
new HondaCar("Honda", "1000cc", 4).describeHondaCar();
```

here countInstances is a static class variable

```
class StaticTest {  
    static countInstances: number = 0;
```

```
    constructor() {  
        StaticTest.countInstances++;
```

```
    }  
}
```

```
new StaticTest();  
new StaticTest();  
console.log(StaticTest.countInstances);
```

TypeScript Notes for Professionals

Traducido por:

rortegag

80+ páginas

de consejos y trucos profesionales

Contenidos

Acerca de	1
Capítulo 1: Introducción a TypeScript.....	2
Sección 1.1: Instalación y configuración.....	2
Sección 1.2: Sintaxis básica.....	4
Sección 1.3: Hola Mundo	5
Sección 1.4: Ejecutar TypeScript con ts-node.....	5
Sección 1.5: TypeScript REPL en Node.js.....	6
Capítulo 2: Por qué y cuándo utilizar TypeScript.....	7
Sección 2.1: Seguridad.....	7
Sección 2.2: Legibilidad.....	7
Sección 2.3: Herramientas	7
Capítulo 3: Tipos principales de TypeScript	8
Sección 3.1: Tipos literales de cadenas de caracteres.....	8
Sección 3.2: Tupla.....	11
Sección 3.3: Booleano.....	11
Sección 3.4: Tipos de intersección.....	11
Sección 3.5: Tipos de argumentos de función y valor de retorno. Number.....	12
Sección 3.6: Tipos de argumentos de función y valor de retorno. String.....	12
Sección 3.7: const Enum.....	13
Sección 3.8: Number.....	14
Sección 3.9: String.....	14
Sección 3.10: Array.....	14
Sección 3.11: Enum.....	14
Sección 3.12: Any.....	15
Sección 3.13: Void.....	15
Capítulo 4: Arrays.....	16
Sección 4.1: Encontrar un objeto en un array.....	16
Capítulo 5: Enums	17
Sección 5.1: Enums con valores explícitos.....	17
Sección 5.2: Cómo obtener todos los valores de enum.....	18
Sección 5.3: Ampliación de enums sin implementación.....	18
Sección 5.4: Implementación de enum personalizada: extiende para enums.....	18
Capítulo 6: Funciones.....	20
Sección 6.1: Parámetros opcionales y por defecto.....	20
Sección 6.2: Función como parámetro.....	20
Sección 6.3: Funciones con tipos de unión	21
Sección 6.4: Tipos de funciones.....	22
Capítulo 7: Clases.....	23

Sección 7.1: Clases abstractas.....	23
Sección 7.2: Clase simple	24
Sección 7.3: Herencia básica	24
Sección 7.4: Constructores.....	24
Sección 7.5: Accesos.....	25
Sección 7.6: Transpilación	26
Sección 7.7: Parchear una función en una clase existente	27
Capítulo 8: Clases.....	28
Sección 8.1: Generación de metadatos mediante un decorador de clases.....	28
Sección 8.2: Pasar argumentos a un decorador de clase.....	29
Sección 8.3: Decorador de clase básico	29
Capítulo 9: Interfaces	31
Sección 9.1: Ampliar la interfaz	31
Sección 9.2: Interfaz de clase.....	31
Sección 9.3: Uso de interfaces para el polimorfismo	32
Sección 9.5: Añadir funciones o propiedades a una existente	33
Sección 9.6: Implantación implícita y forma del objeto	34
Sección 9.7: Uso de interfaces para reforzar tipos	34
Capítulo 10: Genéricos	36
Sección 10.1: Interfaces genéricas.....	36
Sección 10.2: Clase genérica.....	37
Sección 10.3: Parámetros de tipo como limitaciones.....	37
Sección 10.4: Requisitos genéricos	37
Sección 10.5: Funciones genéricas.....	38
Sección 10.6: Uso de Clases y Funciones genéricas:	38
Capítulo 11: Comprobación estricta de nulos	39
Sección 11.1: Comprobación estricta de nulos en acción	39
Sección 11.2: Aserciones no nulas	39
Capítulo 12: Protecciones de tipo definidas por el usuario	40
Sección 12.1: Funciones de protección de tipos.....	40
Sección 12.2: Uso de instanceof	40
Sección 12.3: Uso de typeof	41
Capítulo 13: Ejemplos básicos de TypeScript	42
Sección 13.1: Ejemplo básico de herencia de clases usando extends y la palabra clave super.....	42
Sección 13.2: Ejemplo de variable de clase estática - contar cuántas veces se invoca un método	42
Capítulo 14: Importar bibliotecas externas.....	43
Sección 14.1: Buscar archivos de definición	43
Sección 14.2: Importar un módulo desde npm.....	43
Sección 14.3: Uso de bibliotecas externas globales sin tipado	44
Sección 14.4: Encontrar archivos de definición con TypeScript 2.x.....	44

Capítulo 15: Módulos - exportación e importación	45
Sección 15.1: Módulo Hola mundo	45
Sección 15.2: Reexportar	45
Sección 15.3: Exportar/Importar declaraciones	46
Capítulo 16: Publicar la definición de TypeScript archivos.....	48
Sección 16.1: Incluir archivo de definición con biblioteca en npm.....	48
Capítulo 17: Usar TypeScript con webpack.....	49
Sección 17.1: webpack.config.js	49
Capítulo 18: Mixins	50
Sección 18.1: Ejemplo de Mixins.....	50
Capítulo 19: Cómo utilizar una biblioteca sin un archivo de definición de tipos	51
Sección 19.1: Crear un módulo que exporte un valor por defecto cualquiera.....	51
Sección 19.2: Declarar un any global	51
Sección 19.3: Utilizar un módulo ambiental	51
Capítulo 20: Instalar Typescript y ejecutar el compilador Typescript tsc.....	53
Sección 20.1: Pasos.....	53
Capítulo 21: Configurar el proyecto Typescript para compilar todos los archivos en Typescript.....	55
Sección 21.1: Configuración del archivo TypeScript	55
Capítulo 22: Integración con herramientas de compilación.....	57
Sección 22.1: Browserify.....	57
Sección 22.2: Webpack.....	57
Sección 22.3: Grunt	58
Sección 22.4: Gulp.....	58
Sección 22.5: MSBuild	59
Sección 22.6: NuGet	60
Sección 22.7: Instalar y configurar webpack + loaders	60
Capítulo 23: Uso de TypeScript con RequireJS.....	61
Sección 23.1: Ejemplo de HTML usando RequireJS CDN para incluir un archivo TypeScript ya compilado	61
Sección 23.2: tsconfig.json ejemplo para compilar para ver carpeta usando el estilo de importación RequireJS	61
Capítulo 24: TypeScript con Angular JS.....	62
Sección 24.1: Directiva.....	62
Sección 24.2: Ejemplo simple	63
Sección 24.3: Componente	63
Capítulo 25: TypeScript con SystemJS	65
Sección 25.1: Hola Mundo en el navegador con SystemJS.....	65
Capítulo 26: Uso de TypeScript con React (JS y nativo)	68
Sección 26.1: Componente ReactJS escrito en TypeScript.....	68
Sección 26.2: TypeScript y react y webpack.....	69

Capítulo 27: TSLint: garantizar la calidad del código y coherencia del código	71
Sección 27.1: Configuración para reducir los errores de programación.....	71
Sección 27.2: Instalación y configuración	72
Sección 27.3: Conjuntos de normas TSLint	72
Sección 27.4: Configuración básica de tslint.json.....	72
Sección 27.5: Utilizar un conjunto de reglas predefinido por defecto.....	72
Capítulo 28: tsconfig.json	74
Sección 28.1: Crear proyecto TypeScript con tsconfig.json	74
Sección 28.2: Configuración para reducir los errores de programación.....	75
Sección 28.3: compileOnSave	76
Sección 28.4: Comentarios.....	76
Sección 28.5: preserveConstEnums.....	76
Capítulo 29: Depuración	78
Sección 29.1: TypeScript con ts-node en WebStorm.....	78
Sección 29.2: TypeScript con ts-node en Visual Studio Code	79
Sección 29.3: JavaScript con SourceMaps en Visual Studio Código.....	79
Sección 29.4: JavaScript con SourceMaps en WebStorm	80
Capítulo 30: Pruebas unitarias.....	81
Sección 30.1: tape.....	81
Sección 30.2: jest (ts-jest).....	81
Sección 30.3: Alsatian	83
Sección 30.4: plugin chai-immutable.....	84
Créditos	85

Acerca de

Este libro ha sido traducido por rortegag.com

Si desea descargar el libro original, puede descargarlo desde:

<https://goalkicker.com/TypeScriptBook2/>

Si desea contribuir con una donación, hazlo desde:

<https://www.buymeacoffee.com/GoalKickerBooks>

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/TypeScriptBook2/>

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

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

Envíe sus comentarios y correcciones a web@petercv.com

Capítulo 1: Introducción a TypeScript

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

Sección 1.1: Instalación y configuración

Contexto

TypeScript es un superconjunto tipado de JavaScript que se compila directamente en 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](#).

Visual Studio Code

- [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](#).

Sublime Text

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

Instalación de 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

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 my-code.ts
```

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

Compilar con `tsconfig.json`

También puede proporcionar opciones de compilación que viajen 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. 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 le 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: number = 5;
num = "this is a string"; // error: Type 'string' is not assignable to type 'number'.

```

Los tipos básicos son:

- `number` (tanto enteros como de coma flotante)
- `string`
- `boolean`
- `Array`. Puede especificar los tipos de los elementos de un array. Existen dos formas equivalentes de definir tipos de array: `Array<T>` y `T[]`. Por ejemplo:
 - `number[]` - array de números
 - `Array<string>` - array de cadenas de caracteres
- `Tuplas`. Las tuplas tienen un número fijo de elementos con tipos específicos.
 - `[boolean, string]` - tupla donde el primer elemento es un booleano y el segundo es una cadena de caracteres.
 - `[number, number, number]` - tupla de tres números.
- `{}` - objeto, puede definir sus propiedades o indexador
 - `{name: string, age: number}` - objeto con atributos de nombre y edad
 - `{[key: string]: number}` - diccionario de números indexados por cadena de caracteres
- `enum` - `{ Red = 0, Blue, Green }` - enumeración asignada a números
- `Función`. Se especifican los tipos para los parámetros y el valor de retorno:
 - `(param: number) => string` - función con un parámetro numérico que devuelve cadena de caracteres
 - `() => number` - función sin parámetros que devuelve un número.

- o `(a: string, b?: boolean) => void` - función que toma una cadena de caracteres 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 `undefined` forman parte del tipo `void`.
- `never`
 - o `let foo: never;` - Como el tipo de variables bajo guardas de tipo que nunca son verdaderas.
 - o `function error(message: string): never { throw new Error(message); }` - Como el tipo de retorno de funciones que nunca retornan.
- `null` - tipo para el valor `null`. `null` forma parte implícitamente de todos los tipos, a menos que se active la comprobación estricta de `null`.

Casting

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

```
var derived: 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 derived: MyInterface;
(derived as ImplementingClass).someSpecificMethod();
```

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

Clases

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

Sección 1.3: Hola Mundo

```
class Greeter {
  greeting: string;

  constructor(message: string) {
    this.greeting = message;
  }
  greet(): string {
    return this.greeting;
  }
};

let greeter = new Greeter("Hello, world!");
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 de caracteres que queremos que el método `greet` envíe a la consola. 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 con 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 instalarlo.

```
npm install -g typescript
```

Ejecución del script

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

```
ts-node main.ts
// main.ts
console.log("Hello world");
```

Ejemplo de uso

```
$ ts-node main.ts
Hello world
```

Ejecución de REPL

Para ejecutar REPL ejecute el comando `ts-node`

Ejemplo de uso

```
$ ts-node
> const sum = (a, b): number => a + b;
undefined
> sum(2, 2)
4
> .exit
```

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](#)

Instálelo globalmente con

```
npm install -g tsun
```

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

Ejemplo de uso:

```
$ tsun
TSUN : TypeScript Upgraded Node
type in TypeScript expression to evaluate
type :help for commands in repl
$ function multiply(x, y) {
  ..return x * y;
  ..}
undefined
$ multiply(3, 4)
12
```

Capítulo 2: Por qué y cuándo utilizar 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).

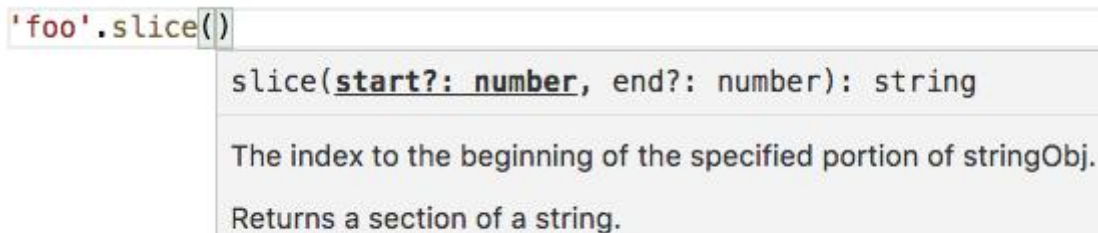
Sección 2.1: Seguridad

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

```
function double(x: number): number {  
    return 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 `(start, stop)` o `(start, length)`.

Sección 2.3: Herramientas

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 principales de TypeScript

Sección 3.1: Tipos literales de cadenas de caracteres

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

```
let myFavoritePet: "dog";
myFavoritePet = "dog";
```

Cualquier otra cadena de caracteres dará error.

```
// Error: El tipo '"rock"' no es assignable al tipo '"dog"'.
// myFavoritePet = "rock";
```

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

```
type Species = "cat" | "dog" | "bird";
```

```
function buyPet(pet: Species, name: string) : Pet { /*...*/ }
```

```
buyPet(myFavoritePet /* "dog", tal como se define más arriba */, "Rocky");
```

```
// Error: Argumento de tipo "rock" no es assignable a parámetro de tipo "cat" | "dog" | "bird". El
tipo "roca" no es assignable al tipo "bird".
// buyPet("rock", "Rocky");
```

Los tipos literales de cadena de caracteres pueden utilizarse para distinguir sobrecargas.

```
function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet { /*...*/ }
```

```
let dog = buyPet(myFavoritePet /* "perro", tal como se define más arriba */, "Rocky");
// dog es de tipo Dog (dog: Dog)
```

Funcionan bien para guardias de tipo definidas por el usuario.

```
interface Pet {
    species: Species;
    eat();
    sleep();
}

interface Cat extends Pet {
    species: "cat";
}

interface Bird extends Pet {
    species: "bird";
    sing();
}

function petIsCat(pet: Pet): pet is Cat {
    return pet.species === "cat";
}

function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}
```

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

Ejemplo de código completo

```
let myFavoritePet: "dog";
myFavoritePet = "dog";

// Error: El tipo "rock" no es asignable al tipo "dog".
// myFavoritePet = "rock";

type Species = "cat" | "dog" | "bird";

interface Pet {
    species: Species;
    name: string;
    eat();
    walk();
    sleep();
}

interface Cat extends Pet {
    species: "cat";
}

interface Dog extends Pet {
    species: "dog";
}

interface Bird extends Pet {
    species: "bird";
    sing();
}

// Error: La interfaz 'Rock' extiende incorrectamente la interfaz 'Pet'. Los tipos de propiedad
// 'species' son incompatibles. El tipo 'rock' no es asignable al tipo '"cat" | "dog" | "bird"'. El
// tipo '"rock"' no es asignable al tipo '"bird"..
// interface Rock extends Pet {
//     type: "rock";
// }

function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet {
    if(pet === "cat") {
        return {
            species: "cat",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
```

```

        console.log(`${this.name} sleeps.`);
    }
    } as Cat;
} else if(pet === "dog") {
    return {
        species: "dog",
        name: name,
        eat: function () {
            console.log(`${this.name} eats.`);
        }, walk: function () {
            console.log(`${this.name} walks.`);
        }, sleep: function () {
            console.log(`${this.name} sleeps.`);
        }
    } as Dog;
} else if(pet === "bird") {
    return {
        species: "bird",
        name: name,
        eat: function () {
            console.log(`${this.name} eats.`);
        }, walk: function () {
            console.log(`${this.name} walks.`);
        }, sleep: function () {
            console.log(`${this.name} sleeps.`);
        }, sing: function () {
            console.log(`${this.name} sings.`);
        }
    } as Bird;
} else {
    throw `Sorry we do not have a ${pet}. Would you like to buy a dog?`;
}
}

function petIsCat(pet: Pet): pet is Cat {
    return pet.species === "cat";
}

function petIsDog(pet: Pet): pet is Dog {
    return pet.species === "dog";
}

function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet) {
    console.log(`Hey ${pet.name}, lets play.`);

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

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

        // Error: El tipo "bird" no es assignable al tipo "cat".
        // pet.type = "bird";

        // Error: La propiedad "sing" no existe en el tipo "Cat".
        // pet.sing();
    } else if(petIsDog(pet)) {
        // pet es ahora de tipo Dog (pet: Dog)
    }
}

```



```

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

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

        pet.eat();
        pet.sing();
        pet.sleep();
    } else {
        throw "An unknown pet. Did you buy a rock?";
    }
}

let dog = buyPet(myFavoritePet /* "dog" tal como se ha definido anteriormente */, "Rocky");
// dog es de tipe Dog (dog: Dog)

// Error: Argumento de tipo "rock" no es asignable a parámetro de tipo "cat" | "dog" | "bird". El
// tipo "rock" no es asignable al tipo "bird"..
// buyPet("rock", "Rocky");

playWithPet(dog);
// Salida: Hey Rocky, lets play.
// Rocky eats.
// Rocky walks.
// Rocky sleeps.

```

Sección 3.2: Tupla

Tipo de array con tipos conocidos y posiblemente diferentes:

```

let day: [number, string];
day = [0, 'Monday']; // valido
day = ['zero', 'Monday']; // invalido: 'zero' no es numerico
console.log(day[0]); // 0
console.log(day[1]); // Monday
day[2] = 'Saturday'; // valid: [0, 'Saturday']
day[3] = false; // invalido: debe ser de tipo union de 'number | string'

```

Sección 3.3: Booleano

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

```

// set with initial value (either true or false)
let isTrue: boolean = true;

// defaults to 'undefined', when not explicitly set
let unsetBool: boolean;

// can also be set to 'null' as well
let nullableBool: boolean = null;

```

Sección 3.4: Tipos de intersección

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

```

interface Knife {
    cut();
}
interface BottleOpener{
    openBottle();
}
interface Screwdriver{
    turnScrew();
}

type SwissArmyKnife = Knife & BottleOpener & Screwdriver;

function use(tool: SwissArmyKnife){
    console.log("I can do anything!");

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

```

Sección 3.5: Tipos de argumentos de función y valor de retorno. Number

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 sum(x: number, y: number): number {
    return x + y;
}

```

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

Uso:

```
sum(84 + 76) // se devolverá 160
```

Nota:

No puede hacerlo

```

function sum(x: string, y: string): number {
    return x + y;
}

```

o

```

function sum(x: number, y: number): string {
    return x + y;
}

```

recibirá los siguientes errores:

error TS2322: Type 'string' is not assignable to type 'number' and error TS2322: Type 'number' is not assignable to type 'string' respectively

Sección 3.6: Tipos de argumentos de función y valor de retorno. String

Ejemplo:

```

function hello(name: string): string {
    return `Hello ${name}!`;
}

```

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

Uso:

```
hello('StackOverflow Documentation') // se devolverá Hello StackOverflow Documentation!
```

Sección 3.7: const Enum

Un `const enum` 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 {  
    Espionage,  
    Sabotage,  
    Assassination  
}
```

// JavaScript: Pero no se genera nada

// TypeScript: Excepto si lo usas

```
let myFavoriteNinjaActivity = NinjaActivity.Espionage;  
console.log(myFavoritePirateActivity); // 0
```

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

```
// var myFavoriteNinjaActivity = 0 /* Espionaje */;
```

```
// console.log(myFavoritePirateActivity); // 0
```

// TypeScript: Lo mismo para el otro ejemplo constante

```
console.log(NinjaActivity["Sabotage"]); // 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 ningún acceso en tiempo de ejecución

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

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

A modo de comparación, un `enum` normal

// TypeScript: Un enum normal

```
enum PirateActivity {  
    Boarding,  
    Drinking,  
    Fencing  
}
```

// JavaScript: El enum después de la compilación

```
// var PirateActivity;
```

```
// (function (PirateActivity) {
```

```
//   PirateActivity[PirateActivity["Boarding"] = 0] = "Boarding";
```

```
//   PirateActivity[PirateActivity["Drinking"] = 1] = "Drinking";
```

```
//   PirateActivity[PirateActivity["Fencing"] = 2] = "Fencing";
```

```
// })(PirateActivity || (PirateActivity = {}));
```

// TypeScript: Un uso normal de este enum

```
let myFavoritePirateActivity = PirateActivity.Boarding;
```

```
console.log(myFavoritePirateActivity); // 0
```

```
// JavaScript: Se parece bastante en JavaScript
// var myFavoritePirateActivity = PirateActivity.Boarding;
// console.log(myFavoritePirateActivity); // 0

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

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

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

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

Sección 3.8: Number

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

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

ECMAScript 2015 permite binario y octal.

```
let binary: number = 0b10; // 2 en decimal
let octal: number = 0o755; // 493 en decimal
```

Sección 3.9: String

Tipo de dato textual:

```
let singleQuotes: string = 'single';
let doubleQuotes: string = "double";
let templateString: string = `I am ${singleQuotes}`; // I am single
```

Sección 3.10: Array

Un array de valores:

```
let threePigs: 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 son 0 por defecto:

```
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
let bestDay: Day = Day.Saturday;
```

Establece un número inicial por defecto:

```
enum TenPlus { Ten = 10, Eleven, Twelve }
```

o asignar valores:

```
enum MyOddSet { Three = 3, Five = 5, Seven = 7, Nine = 9 }
```

Sección 3.12: Any

Si no está seguro del tipo, puede elegir cualquiera:

```
let anything: any = 'I am a string';  
anything = 5; // pero ahora soy el número 5
```

Sección 3.13: Void

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

```
function log(): void {  
    console.log('I return nothing');  
}
```

Tipos **void** Sólo se pueden asignar **null** o **undefined**.

Capítulo 4: Arrays

Sección 4.1: Encontrar un objeto en un array

Uso de `find()`

```
const inventory = [
  {name: 'apples', quantity: 2},
  {name: 'bananas', quantity: 0},
  {name: 'cherries', quantity: 5}
];

function findCherries(fruit) {
  return fruit.name === 'cherries';
}

inventory.find(findCherries); // { name: 'cherries', quantity: 5 }

/* 0 */

inventory.find(e => e.name === 'apples'); // { name: 'apples', 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 de caracteres. Esto podría ser un dolor, pero por suerte existe este método:

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

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

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

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

Puede proporcionar explícitamente valores numéricos utilizando el mismo método

```
enum MyType {
    Value = 3,
    ValueEx = 30,
    ValueEx2 = 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]] = "TwoArr";
    FancyType[FancyType["ThreeArr"] = [3, 3, 3]] = "ThreeArr";
})(FancyType || (FancyType = {}));
```


Sección 5.2: Cómo obtener todos los valores de enum

```
enum SomeEnum { A, B }

let enumValues:Array<string>= [];

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

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

Sección 5.3: Ampliación de enums sin implementación

```
enum SourceEnum {
    value1 = <any>'value1',
    value2 = <any>'value2'
}

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

// necesitamos este tipo para que TypeScript resuelva los tipos correctamente
type 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 personalizada: 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(protected value: string) {}

    public toString() {
        return String(this.value);
    }

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

class SourceEnum extends Enum {
    public static value1 = new SourceEnum('value1');
    public static value2 = new SourceEnum('value2');
}

class TestEnum extends SourceEnum {
    public static value3 = new TestEnum('value3');
    public static value4 = new TestEnum('value4');
}

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

let value1 = TestEnum.value1;

console.log(value1 + 'hello');
console.log(value1.toString() === 'value1');
console.log(value1.is('value1'));
console.log(!TestEnum.value3.is(TestEnum.value3));
console.log(check(TestEnum.value2));
// esto funciona, pero quizás su TSLint se quejaría
// ¡atención! no funciona con ===
// use .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) // Invalid
```

Parámetros por defecto

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

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

```
function buildName(firstName: string, 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(): }) {  
    new constructorFunc();  
}  
  
function foo(constructorWithParamsFunc: { new(num: number): }) {  
    new 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:

```
interface INumberConstructor {  
    new(num: number);  
}  
  
function foo(contructorFunc: INumberConstructor) {  
    new contructorFunc(1);  
}
```

Incluso con genéricos:

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

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

```
function foo(func: { (): void }) {  
    func();  
}  
  
function foo(contructorWithParamsFunc: { (num: number): void }) {  
    new contructorWithParamsFunc(1);  
}
```

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

```
interface IFunction {  
    (): void;  
}  
  
function foo(func: IFunction ) {  
    func();  
}
```

O con parámetros:

```
interface INumberFunction {  
    (num: number): string;  
}  
  
function foo(func: INumberFunction ) {  
    func(1);  
}
```

Incluso con genéricos:

```
interface ITFunc<T, U> {  
    (item: T): U;  
}  
  
function foo<T, U>(contructorFunc: 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(hour:number|string, minute:number|string):string{
    return hour+':'+minute;
}
```

```
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(start:number|string):number{
    if(typeof number === 'string'){
        return parseInt(number)+10;
    } else {
        else return number+10;
    }
}
```

Sección 6.4: Tipos de funciones

Funciones con nombre

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

Funciones anónimas

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

Funciones lambda / flecha

```
let multiply = (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 caracteres 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

```
abstract class Machine {
    constructor(public manufacturer: string) {
    }

    // Una clase abstracta puede definir métodos propios o...
    summary(): string {
        return `${this.manufacturer} makes this machine.`;
    }

    // Exigir a las clases herederas que implementen métodos
    abstract moreInfo(): string;
}

class Car extends Machine {
    constructor(manufacturer: string, public position: number, protected speed: number) {
        super(manufacturer);
    }

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

    moreInfo() {
        return `This is a car located at ${this.position} and going ${this.speed}mph!`;
    }
}

let myCar = new Car("Konda", 10, 70);
myCar.move(); // position es ahora 80
console.log(myCar.summary()); // imprime "Konda makes this machine."
console.log(myCar.moreInfo()); // imprime "This is a car located at 80 and going 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 propios.
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 Car {
  public position: number = 0;
  private speed: number = 42;

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

En este ejemplo, declaramos una clase simple `Car`. 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();           // create an instance of Car
car.move();                    // call a method
console.log(car.position);     // access a public property
```

Sección 7.3: Herencia básica

```
class Car {
  public position: number = 0;
  protected speed: number = 42;

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

class SelfDrivingCar extends Car {

  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 clase `SelfDrivingCar` anula el método `move()` y utiliza la implementación de la clase base usando `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 llaman *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 Car {
    public position: number;
    protected speed: number;

    constructor(position: number, speed: number) {
        this.position = position;
        this.speed = speed;
    }

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

```

Todo este código puede resumirse en un único constructor:

```

class Car {
    constructor(public position: number, protected speed: 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 Car(position, speed) {
        this.position = position;
        this.speed = speed;
    }
    Car.prototype.move = function () {
        this.position += this.speed;
    };
    return Car;
})();

```

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);
        if (startAutoPilot) {
            this.move();
        }
    }
}

let car = new SelfDrivingCar(true);
console.log(car.position); // acceder al puesto de propiedad pública

```

Sección 7.5: Accesos

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

```

class Car {
    public position: number = 0;
    private _speed: number = 42;
    private _MAX_SPEED = 100

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

    get speed(): number {
        return this._speed;
    }

    set speed(value: number) {
        this._speed = Math.min(value, this._MAX_SPEED);
    }
}

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

```

Sección 7.6: Transpilación

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

Fuente TypeScript

```

class SomeClass {

    public static SomeStaticValue: string = "hello";
    public someMemberValue: number = 15;
    private somePrivateValue: boolean = false;

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

    public static getGoodbye(): string {
        return "goodbye!";
    }

    public getFortyTwo(): number {
        return 42;
    }

    private getTrue(): boolean {
        return 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();  
    }  
    SomeClass.getGoodbye = function () {  
        return "goodbye!";  
    };  
    SomeClass.prototype.getFortyTwo = function () {  
        return 42;  
    };  
    SomeClass.prototype.getTrue = function () {  
        return true;  
    };  
    return SomeClass;  
})();  
SomeClass.SomeStaticValue = "hello";
```

Observaciones

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

Sección 7.7: Parchear una función en una clase existente

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

```
interface String {  
    toCamelCase(): string;  
}
```

Ahora podemos parchear esta función en la implementación de `String`.

```
String.prototype.toCamelCase = function() : string {  
    return this.replace(/^[^a-z ]/ig, '')  
        .replace(/(?:^[^w][A-Z]|\b[w]\s+)/g, (match: any, index: number) => {  
            return +match === 0 ? "" : match[index === 0 ? 'toLowerCase' : 'toUpperCase']();  
        });  
}
```

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

```
"This is an example".toCamelCase(); // => "thisIsAnExample"
```

Capítulo 8: Clases

Parámetro

target

Detalles

La clase decorada

Sección 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 metadatos  
    target.__customMetadata = {  
        someKey: "someValue"  
    };  
  
    // devuelve target  
    return target;  
}
```

A continuación, podemos aplicar el decorador de clase:

```
@addMetadata  
class Person {  
    private _name: string;  
    public constructor(name: string) {  
        this._name = name;  
    }  
    public greet() {  
        return this._name;  
    }  
}  
  
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 person1 = new Person("John");  
let person2 = new Person("Lisa");  
  
console.log(getMetadataFromInstance(person1));  
console.log(getMetadataFromInstance(person2));
```

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;  
  
        // devuelve target  
        return target;  
    }  
}
```

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 Person {  
    private _name: string;  
    public constructor(name: string) {  
        this._name = name;  
    }  
    public greet() {  
        return this._name;  
    }  
}
```

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

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

```
console.log(getMetadataFromClass(Person));
```

Si todo ha ido bien debería aparecer la consola:

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

Sección 8.3: Decorador de clase básico

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>(target: T) {  
  
    // Hacer algo con target  
    console.log(target);  
  
    // devuelve target  
    return target;  
}
```

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

```
@log
class Person {
  private _name: string;
  public constructor(name: string) {
    this._name = name;
  }
  public greet() {
    return this._name;
  }
}
```

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 interfaz.

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

Sección 9.1: Ampliar la interfaz

Supongamos que tenemos una interfaz:

```
interface IPerson {  
    name: string;  
    age: number;  
  
    breath(): void;  
}
```

Y queremos crear una interfaz más específica que tenga las mismas propiedades de la persona, podemos hacerlo usando la palabra clave `extends`:

```
interface IManager extends IPerson {  
    managerId: number;  
  
    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.

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

Aquí creamos una clase que implementa la interfaz.

```
class SampleClass implements ISampleClassInterface {  
    public sampleVariable: string;  
    private answerToLifeTheUniverseAndEverything: number;  
  
    constructor() {  
        this.sampleVariable = 'string value';  
        this.answerToLifeTheUniverseAndEverything = 42;  
    }  
    public sampleMethod(): void {  
        // do nothing  
    }  
    private answer(q: any): number {  
        return this.answerToLifeTheUniverseAndEverything;  
    }  
}
```


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

Sección 9.3: Uso de interfaces para el 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 Connector{
    doConnect(): boolean;
}
```

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

```
export class WifiConnector implements Connector{

    public doConnect(): boolean{
        console.log("Connecting via wifi");
        console.log("Get password");
        console.log("Lease an IP for 24 hours");
        console.log("Connected");
        return true
    }

}
```

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

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

```
export class System {
    constructor(private connector: Connector){ #inject Connector type
        connector.doConnect()
    }
}
```

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

```
export class BluetoothConnector implements Connector{

    public doConnect(): boolean{
        console.log("Connecting via Bluetooth");
        console.log("Pair with PIN");
        console.log("Connected");
        return true
    }

}
```

Ver que Wifi y Bluetooth tienen su propia implementación. Su propia manera diferente de conectarse. Sin embargo, ambos han implementado el tipo `Connector` y ahora son de tipo `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 cualquier otro implementando la interfaz `Connector`.

Esto se llama [Duck typing](#). El tipo `Connector` 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 la interfaz ha 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

```
interface IStatus<U> {
    code: U;
}

interface IEvents<T> {
    list: T[];
    emit(event: 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 State<T> implements IEvents<T> {
    list: T[];

    constructor() {
        this.list = [];
    }

    emit(event: T): void {
        this.list.push(event);
    }

    GetAll(): T[] {
        return this.list;
    }
}
```

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

En nuestro ejemplo, la clase `State` manejará un estado genérico utilizando `IStatus<T>`. De esta forma, la interfaz `IEvent<T>` también manejará 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(event => console.log(event.code));
```

Aquí nuestra clase `State` está tipada como `IStatus<number>`.

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

//We are able to emit code as the type Code
s2.emit({ code: { message: 'OK', status: 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 forma, podemos pasar tipos más complejos a nuestro método `emit`.

Como puede 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 extenderlo fácilmente declarando las funciones añadidas por el plugin en una declaración de interfaz separada con el mismo nombre de `JQuery`:

```
interface JQuery {
    pluginFunctionThatDoesNothing(): void;

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

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

Sección 9.6: Implantación implícita y forma del objeto

TypeScript admite interfaces, pero el compilador genera JavaScript, que no las admite. 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.

```
interface IKickable {
    kick(distance: number): void;
}

class Ball {
    kick(distance: number): void {
        console.log("Kicked", distance, "meters!");
    }
}

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

Así, incluso si `Ball` no implementa explícitamente `IKickable`, una instancia de `Ball` puede ser asignada a (y manipulada como) un `IKickable`, incluso cuando se especifica el tipo.

Sección 9.7: Uso de interfaces para imponer tipos

Uno de los principales beneficios de TypeScript es que refuerza 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(petOne, petTwo) {  
    if (petOne.species === petTwo.species && Math.abs(petOne.age - petTwo.age) <= 5) {  
        return 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 saber que se supone que deben pasarle objetos con las propiedades "species" y "age". 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: {species: string, age: number}, petTwo: {species: string, age: number})  
{  
    //...  
}
```

En este caso, 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:

```
interface Pet {  
    species: string;  
    age: number;  
    // 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: Pet, petTwo: Pet) {  
    //...  
}
```

... ¡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

```
interface IResult<T> {
    wasSuccessful: boolean;
    error: T;
}

var result: IResult<string> = ....
var error: string = result.error;
```

Interfaz genérica con múltiples parámetros de tipo

```
interface IRunnable<T, U> {
    run(input: T): U;
}

var runnable: IRunnable<string, number> = ...
var input: string;
var result: number = runnable.run(input);
```

Implementación de una interfaz genérica

```
interface IResult<T>{
    wasSuccessful: boolean;
    error: T;

    clone(): IResult<T>;
}
```

Impleméntalo con una clase genérica:

```
class Result<T> implements IResult<T> {
    constructor(public result: boolean, public error: T) {
    }

    public clone(): IResult<T> {
        return new Result<T>(this.result, this.error);
    }
}
```

Impleméntalo 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 Result<T> {
    constructor(public wasSuccessful: boolean, public error: T) {
    }

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

let r1 = new Result(false, 'error: 42'); // Compiler infers T to string
let r2 = new Result(false, 42); // Compiler infers T to number
let r3 = new Result<string>(true, null); // Explicitly set T to string
let r4 = new Result<string>(true, 4); // Compilation error because 4 is not a string
```

Sección 10.3: Parámetros de tipo como limitaciones

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];
    }
    return target;
}

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

Sección 10.4: Requisitos genéricos

Restricción simple:

```
interface IRunnable {
    run(): void;
}

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

Restricción más compleja:

```
interface IRunnable<U> {
    run(): U;
}

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

Aún más complejo:

```
interface IRunnable<V> {
    run(parameter: U): V;
}

interface IRunner<T extends IRunnable<U, V>, U, V> {
    runSafe(runnable: T, parameter: U): V;
}
```

Restricciones de tipo en línea:

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

Sección 10.5: Funciones genéricas

En interfaces:

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

En las clases:

```
class Runner implements IRunner {  
    public runSafe<T extends IRunnable>(runnable: T): void {  
        try {  
            runnable.run();  
        } catch(e) {  
        }  
    }  
}
```

Funciones sencillas:

```
function runSafe<T extends IRunnable>(runnable: T): void {  
    try {  
        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);
```

```
// Especifique los tipos genéricos:
```

```
runSafe<Runnable<string>, string>(stringRunnable);
```

```
// 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: Element) {  
    return x.id;  
}  
getId(null); // TypeScript no se queja, pero se trata de un error en tiempo de ejecución.
```

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

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

Debe permitir valores nulos explícitamente:

```
function getId(x: Element|null) {  
    return x.id; // error TS2531: El objeto es posiblemente 'null'.  
}  
getId(null);
```

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

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

Sección 11.2: Aserciones no nulas

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

```
type ListNode = { data: number; 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 deducir esto en la siguiente línea de código:  
    // 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  
    node.next!.data = value;  
}
```


Capítulo 12: Protecciones de tipo definidas por el usuario

Sección 12.1: Funciones de protección de tipos

Puedes declarar funciones que sirvan como guardias de tipo utilizando cualquier lógica que desees.

Toman la forma:

```
function functionName(variableName: any): variableName is DesiredType {  
    // body that returns 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 is string {  
    return typeof test === "string";  
}  
  
function example(foo: any) {  
    if (isString(foo)) {  
        // foo se escribe como cadena de caracteres en este bloque  
        console.log("it's a string: " + foo);  
    } else {  
        // foo es del tipo any en este bloque  
        console.log("don't know what this is! [" + foo + "]");  
    }  
}
```

```
example("hello world"); // imprime "it's a string: hello world"  
example({ something: "else" }); // imprime "don't know what this is! [[object Object]]"
```

El predicado de tipo de una función guardia (el `foo is 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 el 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 caracteres de versión.

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

Sección 12.2: Uso de instanceof

`instanceof` requiere que la variable sea de tipo `any`.

Este código ([pruébalo](#)):

```
class Pet { }
class Dog extends Pet {
    bark() {
        console.log("woof");
    }
}
class Cat extends Pet {
    purr() {
        console.log("meow");
    }
}

function example(foo: any) {
    if (foo instanceof Dog) {
        // foo es del tipo Dog en este bloque
        foo.bark();
    }

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

example(new Dog());
example(new Cat());
```

imprime

woof
meow

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 de caracteres 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 escribirse como `number | string` sin problemas.

Este código ([pruébalo](#)):

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

    if (typeof foo === "string") {
        // foo es del tipo string en este bloque
        console.log("not a number: " + foo);
    }
}

example(23);
example("foo");
```

imprime

123
not a number: foo

Capítulo 13: Ejemplos básicos de TypeScript

Sección 13.1: Ejemplo básico de herencia de clases usando extends y la palabra clave super

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

```
class Car{
    name:string;
    engineCapacity:string;

    constructor(name:string,engineCapacity:string){
        this.name = name;
        this.engineCapacity = engineCapacity;
    }

    describeCar(){
        console.log(`${this.name} car comes with ${this.engineCapacity} displacement`);
    }
}

new Car("maruti ciaz","1500cc").describeCar();
```

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

```
class HondaCar extends Car{
    seatingCapacity:number;

    constructor(name:string,engineCapacity:string,seatingCapacity:number){
        super(name,engineCapacity);
        this.seatingCapacity=seatingCapacity;
    }

    describeHondaCar(){
        super.describeCar();
        console.log(`${this} cars comes with seating capacity of ${this.seatingCapacity}`);
    }
}

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

Sección 13.2: Ejemplo de variable de clase estática - contar cuántas veces se invoca un método

Aquí `countInstance` es una variable estática de clase

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

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

Capítulo 14: Importar bibliotecas externas

Sección 14.1: Buscar archivos de definición

Para Typescript 2.x:

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

```
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 la vieja manera:

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 [inicio rapido](#).

```
npm install -global typings
```

Ahora tenemos acceso a las tipificaciones `cli`.

El primer paso es buscar el paquete utilizado por el proyecto

```
typings search lodash
```

NAME	SOURCE	Homepage	DESCRIPTION	VERSIONS
UPDATED				
lodash	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				

A continuación, decida qué fuente debe instalar desde. Utiliza `dt`, que significa [DefinitelyTyped](#), un repositorio de GitHub donde la comunidad puede editar tipografías.

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 el `typings.json`. Ahora siempre que importemos `loadsh`, `typescript` cargará el archivo de tipado de `lodash`.

Si queremos instalar tipos que sólo se utilizarán para el entorno de desarrollo, podemos suministrar el indicador `--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 abreviada de módulo ambiental* para decirle a TypeScript que un módulo existe cuando no tienes un archivo de definición de tipo para el módulo. TypeScript no será capaz de proporcionar ninguna comprobación tipográfica significativa en este caso.

```
declare module "lodash";
```

// ahora puede importar desde lodash de la forma que desee:

```
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 "someModule";
```

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 de npm [@types](#). 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.

Ej:

```
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

```
// hello.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
function helloES(name: string){
    console.log(`Hola ${name}!`);
}
export {helloES};
export default hello;
```

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).

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

Ejemplo de uso de módulos definidos

```
import {hello, helloES} from "./hello"; // cargar los elementos especificados
import defaultHello from "./hello"; // cargar exportación por defecto en nombre defaultHello
import * as Bundle from "./hello"; // cargar todas las exportaciones como Bundle
import {welcome} from "./welcome"; // nota index.ts se omite
hello("World"); // Hello World!
helloES("Mundo"); // Hola Mundo!
defaultHello("World"); // Hello World!
Bundle.hello("World"); // Hello World!
Bundle.helloES("Mundo"); // Hola Mundo!
welcome("Human"); // Welcome Human!
```

Sección 15.2: Reexportar

TypeScript permite reexportar declaraciones.

```
// Operator.ts
interface Operator {
    eval(a: number, b: number): number;
}
export default Operator;

// Add.ts
import Operator from "./Operator";
export class Add implements Operator {
    eval(a: number, b: number): number {
        return a + b;
    }
}

// Mul.ts
import Operator from "./Operator";
export class Mul implements Operator {
    eval(a: number, b: number): number {
        return 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";
```

También se pueden exportar **exportaciones por defecto**, pero no se dispone de una sintaxis abreviada. Recuerde que sólo es posible una exportación por defecto por módulo.

```
// Calculator.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
import Operator from "./Operator";

export default Operator;
```

Es posible reexportar la **importación agrupada**

```
// RepackedCalculator.ts
export * from "./Operators";
```

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

```
// FixedCalculator.ts
export * from "./Calculator"
import Operator from "./Calculator";
export class Add implements Operator {
    eval(a: number, b: number): number {
        return 42;
    }
}
```

Ejemplo de uso

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

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

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

Sección 15.3: Exportar/Importar declaraciones

Cualquier declaración (variable, constante, 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 hello(name: string){
    console.log(`Hello ${name}!`);
}

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); // Hello 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 hello(name: string){  
    console.log(`Hello ${name}!`);  
}  
export const answerToLifeTheUniverseAndEverything = 42;  
  
import * as Bundle from "./adams";  
Bundle.hello(Bundle.answerToLifeTheUniverseAndEverything); // Hello 42!  
console.log(Bundle.unused); // 0
```


Capítulo 16: Publicar la definición de TypeScript archivos

Sección 16.1: Incluir archivo de definición con biblioteca en npm

Añada tipos a su `package.json`

```
{  
  ...  
  "typings": "path/file.d.ts"  
  ...  
}
```

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

Capítulo 17: Usar TypeScript con webpack

Sección 17.1: webpack.config.js

Instalar cargadores npm `install --save-dev ts-loader source-map-loader`

tsconfig.json

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

module.exports = {
  entry: "./src/index.ts",
  output: {
    filename: "./dist/bundle.js",
  },

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

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

  module: {
    loaders: [
      // Todos los archivos con extensión '.ts' o '.tsx' serán gestionados por 'ts-loader'.
      {test: /\.tsx?$/, loader: "ts-loader"}
    ],

    preLoaders: [
      // Todos los archivos '.js' de salida serán reprocesados por 'source-map-loader'..
      {test: /\.js$/, loader: "source-map-loader"}
    ]
  },
  /**
   * Si desea utilizar react *
   */
  // Al importar un módulo cuya ruta coincida con una de las siguientes, simplemente
  // asumir que existe una variable global correspondiente y usarla en su lugar.
  // Esto es importante porque nos permite evitar la agrupación de todos nuestros
  // dependencias, lo que permite a los navegadores cachear esas librerías entre
  // construcciones.
  // externals: {
  //   "react": "React",
  //   "react-dom": "ReactDOM"
  // },
};
```

Capítulo 18: Mixins

Parámetro	Descripción
<code>derivedCtor</code>	La clase que desea utilizar como clase de composición
<code>baseCtors</code>	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 Flies {
  fly() {
    alert('Is it a bird? Is it a plane?');
  }
}

class Climbs {
  climb() {
    alert('My spider-sense is tingling.');
```

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

```
class BeetleGuy implements Climbs, Bulletproof {
  climb: () => void;
  deflect: () => void;
}

applyMixins(BeetleGuy, [Climbs, Bulletproof]);
```

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(name => {
      if (name !== 'constructor') {
        derivedCtor.prototype[name] = baseCtor.prototype[name];
      }
    });
  });
}
```

Capítulo 19: Cómo utilizar una biblioteca 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 cualquiera

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

Utilizando como ejemplo JQuery (aunque dispone de tipados):

```
// colocar en 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;
    let jQuery: any;

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

A continuación, puedes importar y utilizar ambos nombres:

```
// algún otro archivo .ts
import { $, jQuery } from "jquery";
$.doThing();
jQuery.doOtherThing();
```

Sección 19.2: Declarar un any global

A veces es más fácil declarar un global de tipo `any`, especialmente en proyectos sencillos.

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)
declare module "jquery"; // tenga en cuenta que no hay exportaciones definidas
```

A continuación, puede importar desde el módulo ambiente.

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

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

Capítulo 20: Instalar Typescript y ejecutar el compilador Typescript tsc

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

Sección 20.1: Pasos

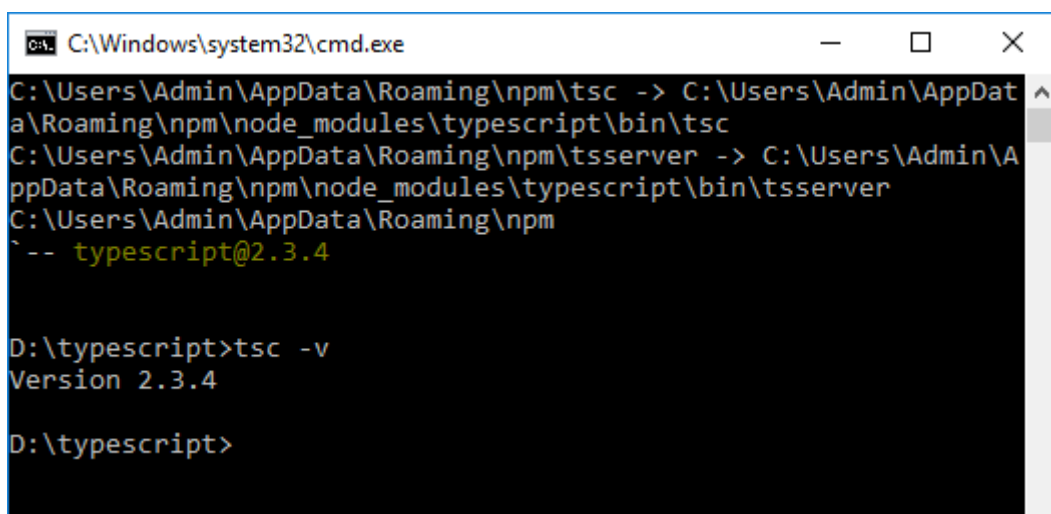
Instalación de TypeScript y ejecución del compilador de TypeScript

Para instalar el compilador 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

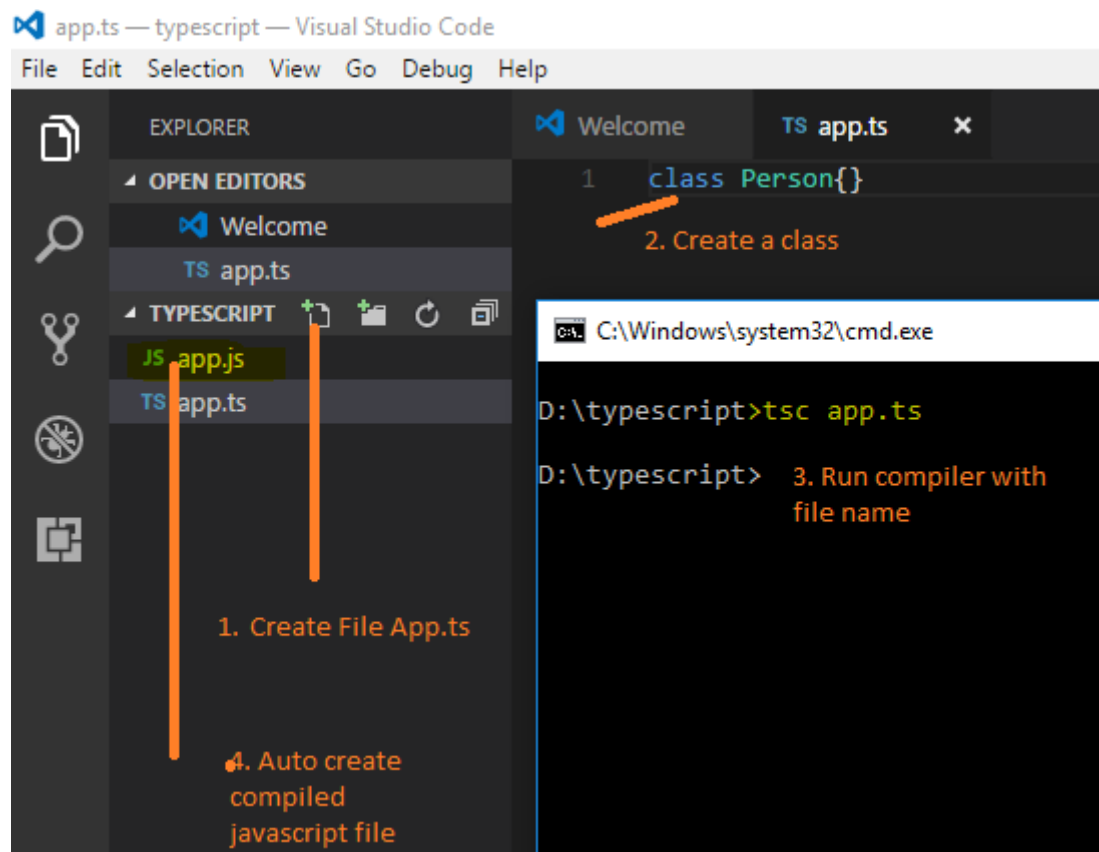
D:\typescript>tsc -v
Version 2.3.4

D:\typescript>
```

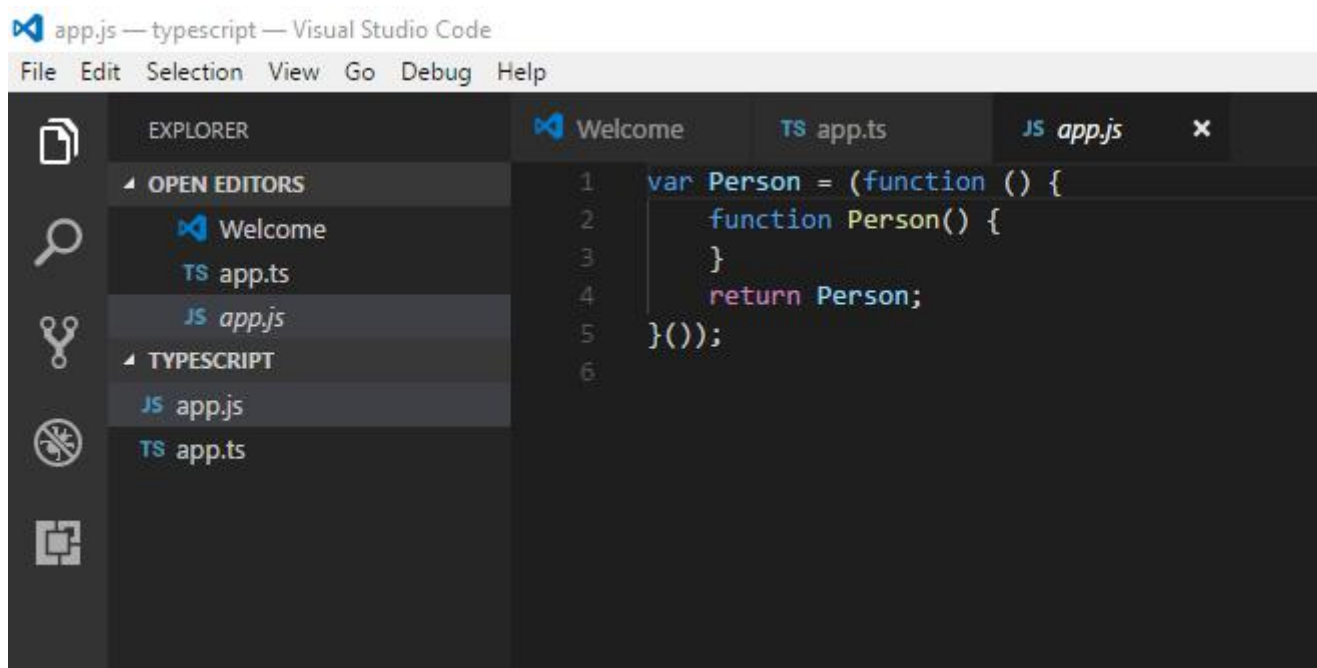
Descargar Visual Studio Code para Linux/Windows

[Enlace de descarga de Visual Code](#)

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



Vea el resultado en javascript compilado del código typescript escrito.



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 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 siguiente archivo de configuración automáticamente.

The screenshot shows the Visual Studio Code interface with the `tsconfig.json` file open in the editor. The file contains the following configuration:

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",
    "module": "commonjs",
    "strict": true
  }
}
```

Below the editor, the Command Prompt shows the execution of the `tsc --init` command, which successfully creates the `tsconfig.json` file. A red box highlights the message: "config file will be auto created to directory".

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

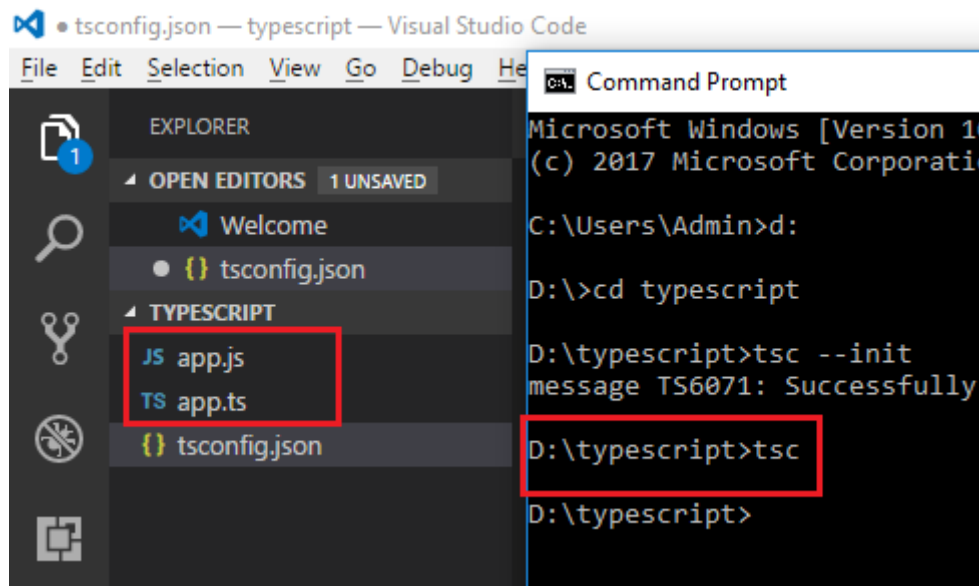
C:\Users\Admin>d:

D:\>cd typescript

D:\typescript>tsc --init
message TS6071: Successfully created a tsconfig.json file.

D:\typescript>
```

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



- 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.

Capítulo 22: Integración con herramientas de compilación

Sección 22.1: Browserify

Instalar

```
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(process.stdout);
```

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

Sección 22.2: Webpack

Instalar

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

webpack.config.js básico

webpack 2.x, 3.x

```
module.exports = {
  resolve: {
    extensions: ['.ts', '.tsx', '.js']
  },
  module: {
    rules: [
      {
        // Configurar ts-loader para archivos .ts/.tsx y excluir cualquier
        // importación de node_modules.
        test: /\.tsx?$/,
        loaders: ['ts-loader'],
        exclude: /node_modules/
      }
    ]
  },
  entry: [
    // Establecer index.tsx como punto de entrada de la aplicación.
    './index.tsx'
  ],
  output: {
    filename: "bundle.js"
  }
};
```

webpack 1.x

```
module.exports = {
  entry: "./src/index.tsx",
  output: {
    filename: "bundle.js"
  },
  resolve: {
    // Añadir '.ts' y '.tsx' como extensión resoluble.
    extensions: [".", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
  },
  module: {
    loaders: [
      // Todos los archivos con extensión '.ts' o '.tsx' serán tratados por 'ts-loader'.
      { test: /\.ts(x)?$/, loader: "ts-loader", exclude: /node_modules/ }
    ]
  }
}
```

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

Alternativas:

- [awesome-typescript-loader](#)

Sección 22.3: Grunt

Instalar

npm **install** grunt-ts

Gruntfile.js básico

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

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

Sección 22.4: Gulp

Instalar

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 puede añadir y sobrescribir parámetros
});

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

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

Sección 22.5: MSBuild

Actualice el archivo del proyecto para incluir los archivos `Microsoft.TypeScript.Default.props` (en la parte superior) y `Microsoft.TypeScript.targets` (en la parte inferior) instalados localmente:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Incluir accesorios por defecto en la parte inferior -->
  <Import
    Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(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 Condition="'$(Configuration)' == 'Debug'">
    <TypeScriptRemoveComments>>false</TypeScriptRemoveComments>
    <TypeScriptSourceMap>true</TypeScriptSourceMap>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)' == 'Release'">
    <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
    <TypeScriptSourceMap>>false</TypeScriptSourceMap>
  </PropertyGroup>

  <!-- Incluir objetivos por defecto en la parte inferior -->
  <Import
    Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets"
    Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets')"/>
</Project>
```

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

Sección 22.6: NuGet

- Clic derecho -> Administrar paquetes NuGet
- Buscar `Microsoft.TypeScript.MSBuild`
- Pulse Install
- Cuando termine la instalación, ¡reconstruya!

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

Sección 22.7: Instalar y configurar webpack + loaders

Instalación

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

webpack.config.js

```
module.exports = {
  entry: {
    app: ['./src/'],
  },
  output: {
    path: __dirname,
    filename: './dist/[name].js',
  },
  resolve: {
    extensions: ['', '.js', '.ts'],
  },
  module: {
    loaders: [{
      test: /\.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 de 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 para ver carpeta usando el estilo de importación RequireJS

```
{
  "module": "amd", // Uso del generador de código de módulos AMD que funciona con RequireJS
  "rootDir": "./src", // Cambie esto a su carpeta de origen
  "outDir": "./view",
  ...
}
```

Capítulo 24: TypeScript con AngularJS

Nombre	Descripción
<code>controllerAs</code>	es un nombre de alias, al que se pueden asignar variables o funciones. @ver: https://docs.angularjs.org/guide/directive
<code>\$inject</code>	Lista de inyección de dependencia, es resuelto por angular y pasar como argumento a las funciones del constructor.

Sección 24.1: Directiva

```
interface IMyDirectiveController {
    // especifique aquí los métodos y propiedades expuestos del controlador
    getUrl(): string;
}

class MyDirectiveController implements IMyDirectiveController {
    // Inyecciones internas, por cada directiva
    public static $inject = ['$location', 'toaster'];

    constructor(private $location: ng.ILocationService, private toaster: any) {
        // $location y toaster son ahora propiedades del controlador
    }

    public getUrl(): string {
        return this.$location.url(); // utilizar $location para recuperar la URL
    }
}

/*
 * Inyecciones exteriores, para controlar una vez la carrera.
 * Por ejemplo tenemos todas las plantillas en un valor, y queremos usarlo.
 */
export function myDirective(templatesUrl: ITemplates): ng.IDirective {
    return {
        controller: MyDirectiveController,
        controllerAs: 'vm',
        link: (scope: ng.IScope,
            element: ng.IAugmentedJQuery,
            attributes: ng.IAttributes,
            controller: IMyDirectiveController): void => {
            let url = controller.getUrl();
            element.text('Current URL: ' + 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
myDirective.prototype.slug = 'myDirective';

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

Sección 24.2: Ejemplo simple

```
export function myDirective($location: ng.ILocationService): ng.IDirective {
  return {
    link: (scope: ng.IScope,
    element: ng.IAugmentedJQuery,
    attributes: ng.IAttributes): void => {
      element.text("Current URL: " + $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 de la
directiva
myDirective.prototype.slug = "myDirective";

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

Sección 24.3: Componente

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

myModule.ts

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

angular.module("myModule", [])
  .component("myModuleComponent", new MyModuleComponent())
  .service("myModuleService", MyModuleService);
```

components/myModuleComponent.ts

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

export class MyModuleComponent implements IComponentOptions {
  public templateUrl: string = "../app/myModule/templates/myComponentTemplate.html";
  public controller: Injectable<IControllerConstructor> = MyModuleController;
  public bindings: {[boundProperty: string]: string} = {};
}
```

templates/myModuleComponent.html

```
<div class="my-module-component">
  {{ctrl.someContent}}
</div>
```


controller/MyModuleController.ts

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

export class MyModuleController implements IController {
    public static readonly $inject: string[] = ["$element", "myModuleService"];
    public someContent: string = "Hello World";

    constructor($element: JQuery, private myModuleService: MyModuleService) {
        console.log("element", $element);
    }

    public doSomething(): void {
        // implementacion..
    }
}
```

services/MyModuleService.ts

```
export class MyModuleService {
    public static readonly $inject: string[] = [];

    constructor() {
    }

    public doSomething(): void {
        // hacer algo
    }
}
```

somewhere.html

```
<my-module-component></my-module-component>
```

Capítulo 25: TypeScript con SystemJS

Sección 25.1: Hola Mundo 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 publicado.

Para TypeScript 1.8 tienes que usar plugin-typescript 4.0.16

Crear archivo hello.ts

```
export function greeter(person: String) {
    return 'Hello, ' + person;
}
```

Crear archivo hello.html

```
<!doctype html>
<html>
  <head>
    <title>Hello World in 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(hello) {
          document.body.innerHTML = hello.greeter('World');
        });
      });
    </script>

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

Crear config.js - Archivo de configuración de SystemJS

```
System.config({
  packages: {
    "plugin-typescript": {
      "main": "plugin.js"
    },
    "typescript": {
      "main": "lib/typescript.js",
      "meta": {
        "lib/typescript.js": {
          "exports": "ts"
        }
      }
    }
  },
  map: {
    "plugin-typescript": "node_modules/plugin-typescript/lib/",
    /* NOTE: this is for npm 3 (node 6) */
    /* for npm 2, typescript path will be */
    /* node_modules/plugin-typescript/node_modules/typescript */
    "typescript": "node_modules/typescript/"
  },
  transpiler: "plugin-typescript",
  meta: {
    "./hello.ts": {
      format: "esm",
      loader: "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 el ejemplo html.

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 `hello-production.html`:

```
<!doctype html>
<html>
  <head>
    <title>Hello World in TypeScript</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

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

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

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

```
ReactDOM.render(<HelloMessage name="John" />, 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:

```
// helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

```
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

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

2. Añadir interfaces Props y State:

```
interface Props {
  name:string;
  optionalParam?:number;
}

interface State {
  // vacío en nuestro caso
}

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

// TypeScript le permitirá crear sin el parámetro opcional
ReactDOM.render(<HelloMessage name="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 y react y 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 = {
  entry: "<path to entry point>", // por ejemplo ./src/helloMessage.tsx
  output: {
    filename: "<path to bundle file>", // por ejemplo ./dist/bundle.js
  },

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

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

  module: {
    loaders: [
      // Todos los archivos con extensión '.ts' o '.tsx' serán gestionados por 'ts-loader'.
      { test: /\.tsx?$/, loader: "ts-loader" }
    ],

    preLoaders: [
      // Todos los archivos '.js' de salida serán reprocesados por 'source-map-loader'.
      { test: /\.js$/, loader: "source-map-loader" }
    ]
  },
}
```

```
// Al importar un módulo cuya ruta coincida con una de las siguientes, simplemente  
// asumir que existe una variable global correspondiente y usarla en su lugar.  
// Esto es importante porque nos permite evitar la agrupación de todos nuestros  
// dependencias, lo que permite a los navegadores cachear esas librerías entre  
construcciones.
```

```
externals: {  
    "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: garantizar la calidad del código y coherencia 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 compilarlo con `tsc`.

```
{
  "rules": {
    // Específico de TypeScript
    "member-access": true,
    // Requiere declaraciones explícitas de visibilidad para los miembros de la clase.
    "no-any": true, // No permite el uso de any como declaración de tipo.
    // Funcionalidad
    "label-position": true, // Sólo permite etiquetas en lugares razonables.
    "no-bitwise": true, // No permite operadores bit a bit.
    "no-eval": true, // No permite invocar funciones eval.
    "no-null-keyword": true, // No permite el uso de la palabra clave literal null.
    "no-unsafe-finally": true,
    // No permite sentencias de flujo de control, como return, continue, break y throws en bloques finally.
    "no-var-keyword": true, // No permite 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,
    // Obliga a utilizar la función isNaN() para comprobar las referencias NaN en lugar de una comparación con la constante NaN.
    // Estilo
    "class-name": true, // Impone el uso de nombres de clase e interfaz en PascalCased.
    "interface-name": [ true, "never-prefix" ],
    // Exige que los nombres de las interfaces empiecen por "I" mayúscula
    "no-angle-bracket-type-assertion": true,
    // Requiere el uso de as Type para las aserciones de tipo en lugar de <Type>.
    "one-variable-per-declaration": true,
    // No permite definir varias variables en la misma declaración.
    "quotemark": [ true, "double", "avoid-escape" ],
    // Requiere comillas dobles para los literales de cadena de caracteres.
    "semicolon": [ true, "always" ],
    // Impone el uso coherente 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. No permite el uso de ciertas palabras clave de TypeScript (any, Number, number, String, string, Boolean, boolean, undefined) como variable o parámetro. Sólo permite nombres de variables en MAYÚSCULAS. Permite guiones bajos al principio (sólo tiene efecto si se especifica "checkformat").
  }
}
```


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 filename.ts
```

Sección 27.3: Conjuntos de normas TSLint

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

El generador Yeoman admite todos estos preajustes y también puede ampliarse:

- [generator-tslint](#)

Sección 27.4: Configuración básica de tslint.json

Esta es una configuración básica `tslint.json` que

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

```
{
  "rules": {
    "no-any": true,
    "curly": true,
    "quotemark": [true, "double"]
  }
}
```

Sección 27.5: Utilizar un conjunto de reglas predefinido por defecto

`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 en las 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`.

[Documentación](#) y [código fuente del conjunto de reglas predefinidas](#)

Así que uno puede simplemente utilizar:

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

para tener una configuración inicial sensata.

A continuación, se pueden sobrescribir las reglas de ese preajuste mediante `rules`, por ejemplo, para los desarrolladores de nodos tenía sentido establecer `no-console` en **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 proyecto habilitado para TypeScript.

Inicializar un proyecto TypeScript, o mejor dicho un archivo **tsconfig.json**, puede hacerse 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", /* Especifique la versión de destino de ECMAScript: 'ES3' (por defecto), 'ES5', 'ES2015', 'ES2016', 'ES2017', o 'ESNEXT'. */
    "module": "commonjs", /* Especificar 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", /* Especificar la generación de código JSX: 'preserve', 'reactnative', 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 archivo. */
    // "outDir": ".", /* Redirigir la estructura de salida al directorio. */
    // "rootDir": ".", /* Especifique el directorio raíz de los archivos de entrada. Utilícelo 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', propagación y desestructuración cuando se apunta a 'ES5' o 'ES3'.. */
    // "isolatedModules": true, /* Transpile cada archivo como un módulo separado (similar a 'ts.transpileModule'). */
    /* Opciones de comprobación tipográfica estricta */
    "strict": true /* Activar todas las opciones de comprobación tipográfica estricta. */
    // "noImplicitAny": true, /* Emitir error en expresiones y declaraciones con un tipo 'any' implícito. */
    // "strictNullChecks": true, /* Activar la comprobación estricta de nulos. */
    // "noImplicitThis": true, /* Error en expresiones "this" con un tipo "any" implícito. */
    // "alwaysStrict": true, /* Parsear en modo estricto y emitir "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 de la función devuelven un valor. */
    // "noFallthroughCasesInSwitch": true, /* Notificación de errores en los casos en los que se ha producido un fallo en la sentencia switch. */
    /* Opciones de resolución del módulo */
    // "moduleResolution": "node", /* Especifique la estrategia de resolución de módulos: 'node' (Node.js) o 'classic' (TypeScript pre-1.6). */
    // "baseUrl": ".", /* Directorio base para resolver nombres de módulos no absolutos. */
    // "paths": {}, /* Una serie de entradas que reasignan las importaciones a ubicaciones de búsqueda 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. */
// "types": [], /* Archivos de declaración de tipos que deben incluirse 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 fuentes */
// "sourceRoot": "./", /* Especifique la ubicación donde el depurador debe localizar los
archivos TypeScript en lugar de las ubicaciones de origen. */
// "mapRoot": "./", /* Especifique la ubicación donde el depurador debe localizar los
archivos de mapa en lugar de las ubicaciones generadas. */
// "inlineSourceMap": true, /* Emitir un único archivo con los mapas de fuentes en lugar
de tener un archivo separado. */
// "inlineSources": true, /* Emitir el código fuente junto con los mapas de fuentes en
un único archivo; es necesario definir "--inlineSourceMap" o "--sourceMap". */
/* Opciones experimentales */
// "experimentalDecorators": true, /* Habilita la compatibilidad experimental con los
decoradores ES7.*/
// "emitDecoratorMetadata": true, /* Habilita la compatibilidad experimental con la
emisión de metadatos de tipo para los decoradores. */
}
}

```

La mayoría de las opciones, si no todas, se generan automáticamente y sólo se deja sin comentar lo estrictamente necesario.

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, // Parsear en modo estricto y emitir "use strict" para cada
    fichero fuente.
    // Si el nombre del archivo es Global.ts y tiene un /// <reference path="global.ts" />
    para hacer referencia a este archivo, puede provocar errores inesperados.
    Visite: http://stackoverflow.com/questions/36628612/typescript-transpiler-casing-issue
    "forceConsistentCasingInFileNames": true, // No permitir referencias a un mismo archivo
    con casillas incoherentes.
    // "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, // Notificación de errores en los casos de "fall
    through" 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, // Error en expresiones y declaraciones con un tipo "any"
    implícito.
  }
}

```

```

    "noImplicitThis": true, // Error en estas expresiones con un tipo "any" implícito.
    "strictNullChecks": true, // Los valores nulos e indefinidos no están en el dominio de
    // Para aplicar estas reglas, añade esta configuración.
    "noEmitOnError": true // No emitir salidas si se ha notificado 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": {
    ...
  },
  "exclude": [
    ...
  ]
}

```

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 Update 3](#)
- [JetBrains WebStorm](#)
- Atom [con atom-typescript](#)

Sección 28.4: Comentarios

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

```

// Comentario principal
{
  "compilerOptions": {
    // este es un comentario de línea
    "module": "commonjs", // eol comentario de línea
    "target" /* bloque en línea */ : "es5",
    /* Se trata de un
    bloque
    comentario */
  }
}
/* comentario final */

```

Sección 28.5: preserveConstEnums

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

Por lo general, esto no es más que azúcar sintáctico, ya que los enums constantes están inlineados en JavaScript compilado.

Por ejemplo, el siguiente código

```
const enum Tristate {  
    True,  
    False,  
    Unknown  
}  
  
var something = Tristate.True;
```

se compila en

```
var something = 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,  
    ...  
  },  
  "exclude": [  
    ...  
  ]  
}
```

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["True"] = 0] = "True";  
    Tristate[Tristate["False"] = 1] = "False";  
    Tristate[Tristate["Unknown"] = 2] = "Unknown";  
})(Tristate || (Tristate = {}));  
  
var something = 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 TypeScript.

o

Ejecute TypeScript directamente con `ts-node`

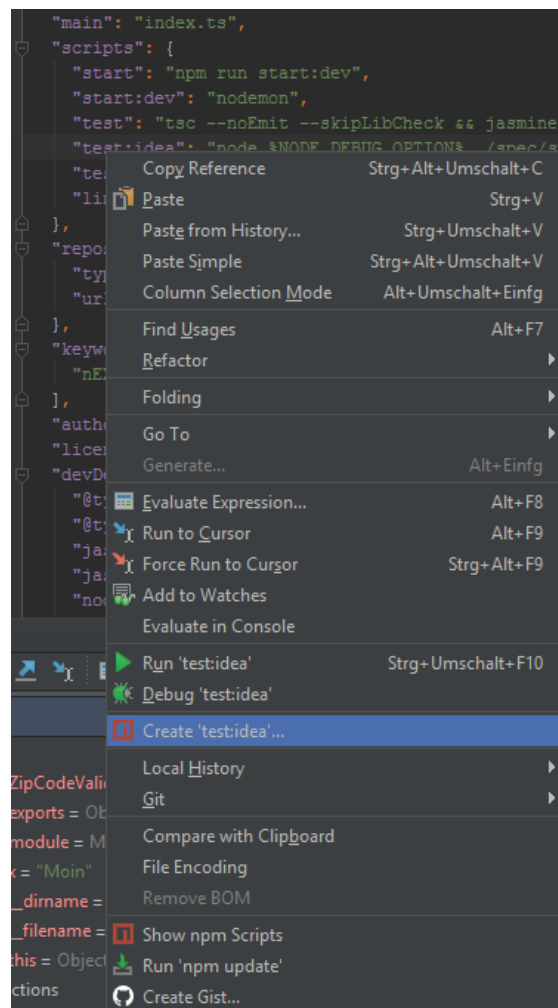
Este artículo describe ambas formas utilizando [Visual Studio Code](#) y [WebStorm](#). Todos los ejemplos suponen que el archivo principal es `index.ts`.

Sección 29.1: TypeScript con ts-node en WebStorm

Añade este script a tu `package.json`:

```
"start: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": [
    {
      "type": "node2",
      "request": "launch",
      "name": "Launch Program",
      "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 Código

En el `tsconfig.json` establezca

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

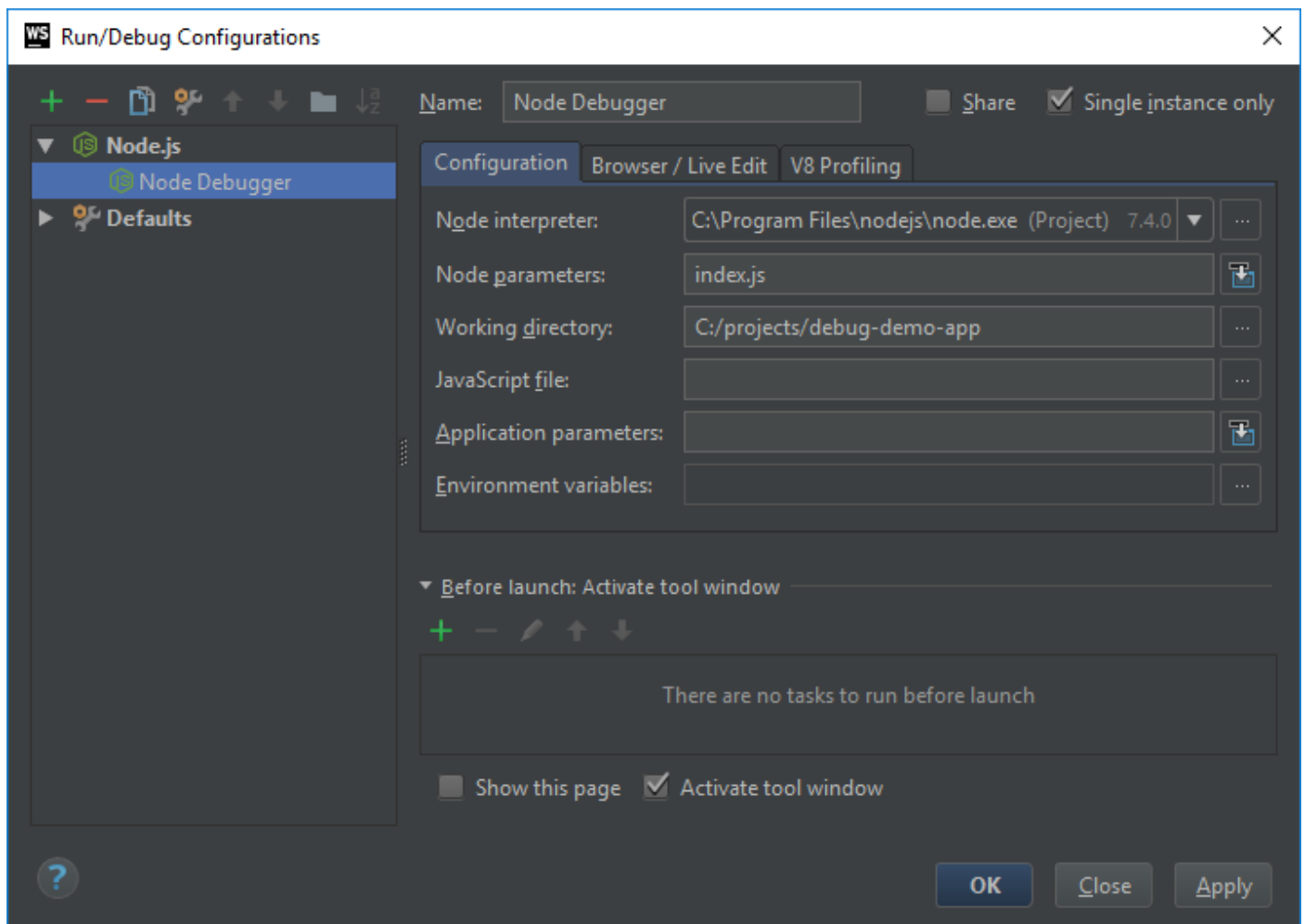
para generar mapeos junto con los archivos js de las fuentes TypeScript utilizando el comando `tsc`. El archivo [launch.json](#):

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "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

Sección 30.1: tape

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

Para instalar `tape` utilizando el comando de ejecución `npm`

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

Para utilizar `tape` 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 "tape";

test("Math test", (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 version 13
# Math test
ok 1 should be equal
ok 2 should be truthy

1..2
# tests 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 `tests` ejecute el comando

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

Sección 30.2: jest (ts-jest)

[jest](#) es un marco de pruebas JavaScript indoloro de Facebook, con [ts-jest](#) se puede utilizar para probar el 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": {
    "transform": {
      "。(ts|tsx)": "<rootDir>/node_modules/ts-jest/preprocessor.js"
    },
    "testRegex": "(/__tests__/.*|\\.(test|spec))\\.。(ts|tsx|js)$",
    "moduleFileExtensions": ["ts", "tsx", "js"]
  }
}
```

Ahora `jest` está listo. Supongamos que tenemos el ejemplo `fizz buzz` 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) {
      output += 'Fizz ';
    }
    if (i % 5 === 0) {
      output += 'Buzz ';
    }
  }
  return output;
}
```

Un ejemplo de prueba podría ser

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

import {fizzBuzz} from "./fizzBuzz";
test("FizzBuzz test", () =>{
  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
✓ FizzBuzz test (3ms)
```

```
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.46s, estimated 2s
Ran all test suites.
```

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 `fizz buzz` muestra debe ver

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

```
-----|-----|-----|-----|-----|-----|
File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Lines |
-----|-----|-----|-----|-----|-----|
All files | 92.31 | 87.5 | 100 | 91.67 | |
fizzBuzz.ts | 92.31 | 87.5 | 100 | 91.67 | 13 |
-----|-----|-----|-----|-----|
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.857s
Ran all test suites.
```

`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	<div><div></div></div> 87.5% 7/8	<div><div></div></div> 100% 1/1	<div><div></div></div> 91.67% 11/12

Sección 30.3: Alsatian

[Alsatian](#) es un framework de pruebas unitarias escrito en TypeScript. Permite el uso de casos de prueba y genera 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 } from "alsatian";
import { SomeModule } from "../src/some-module";

export SomeModuleTests {

    @Test()
    public statusShouldBeTrueByDefault() {
        let instance = new SomeModule();
        Expect(instance.status).toBe(true);
    }

    @Test("Name should be null by default")
    public nameShouldBeNullByDefault() {
        let instance = new SomeModule();
        Expect(instance.name).toBe(null);
    }

    @TestCase("first name")
    @TestCase("apples")
    public shouldSetNameCorrectly(name: string) {
        let instance = new SomeModule();

        instance.setName(name);

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

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

Sección 30.4: plugin chai-immutable

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

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

2. Instala los @types 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 immutable example', () => {
    it('example', () => {
        expect(Set.of(1,2,3)).to.not.be.empty;
        expect(Set.of(1,2,3)).to.include(2);
        expect(Set.of(1,2,3)).to.include(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, más cambios pueden ser enviados a web@petercv.com para que el nuevo contenido sea publicado o actualizado.

Traductor al español

[rortegag](#)

2426021684	Capítulos 1, 14 y 16
ABabin	Capítulo 9
Alec Hansen	Capítulo 1
Alex Filatov	Capítulos 22 y 27
Almond	Capítulo 14
Aminadav	Capítulo 9
Aron	Capítulo 9
artem	Capítulos 9, 14 y 25
Blackus	Capítulo 14
bnieland	Capítulo 28
br4d	Capítulo 6
BrunoLM	Capítulos 1, 17 y 22
Brutus	Capítulo 14
ChanceM	Capítulo 1
Cobus Kruger	Capítulo 9
danvk	Capítulos 1, 2 y 11
dimitrisli	Capítulo 5
dublicator	Capítulo 14
Equiman	Capítulo 7
Fenton	Capítulos 3 y 18
Florian Hämmerle	Capítulo 5
Fylax	Capítulos 1, 3 y 28
goenning	Capítulo 28
hansmaad	Capítulos 7 y 10
Harry	Capítulo 14
irakli khitarishvili	Capítulos 17 y 26
islandman93	Capítulos 1, 6, 9, 14 y 26
James Monger	Capítulos 7, 27 y 30
JKillian	Capítulos 11 y 14
Joel Day	Capítulo 14
John Ruddell	Capítulo 22
Joshua Breeden	Capítulos 1 y 9
Juliën	Capítulos 3 y 28
Justin Niles	Capítulo 7
k0pernikus	Capítulos 1 y 27
Kevin Montrose	Capítulos 5, 12 y 19
Kewin Dousse	Capítulo 22
KnottytOmo	Capítulos 1, 10 y 14
Kuba Beránek	Capítulo 1
Lekhath	Capítulo 1
leonidv	Capítulo 30
lilezek	Capítulo 23
Magu	Capítulos 3, 27 y 28

Matt Lishman	Capítulo 1
Matthew Harwood	Capítulo 30
Mikhail	Capítulos 1 y 3
mleko	Capítulos 1, 15, 22, 27 y 30
muetzerich	Capítulo 6
Muhammad Awais	Capítulo 10
Paul Boutes	Capítulo 9
Peopleware	Capítulo 29
Rahul	Capítulos 20 y 21
Rajab Shakirov	Capítulos 14 y 26
RationalDev	Capítulos 1 y 3
Remo H. Jansen	Capítulo 8
Robin	Capítulo 7
Roman M. Koss	Capítulo 24
Roy Dictus	Capítulo 1
Saiful Azad	Capítulos 1 y 9
Sam	Capítulo 1
samAlvin	Capítulo 1
SilentLupin	Capítulo 6
Slava Shpitalny	Capítulos 6, 9, 10 y 14
smnbbrv	Capítulo 5
Stefan Rein	Capítulo 24
Sunnyok	Capítulo 9
Taytay	Capítulo 10
Udlei Nati	Capítulo 4
user3893988	Capítulo 28
vashishth	Capítulo 13
Wasabi Fan	Capítulo 1