



EBook Gratis

APRENDIZAJE TypeScript

Free unaffiliated eBook created from
Stack Overflow contributors.

#typescript

Tabla de contenido

Acerca de.....	1
Capítulo 1: Empezando con TypeScript.....	2
Observaciones.....	2
Versiones.....	2
Examples.....	3
Instalación y configuración.....	3
Fondo.....	3
IDEs.....	3
Estudio visual.....	3
Código de Visual Studio.....	4
Tormenta web.....	4
IntelliJ IDEA.....	4
Atom & atom-mecanografiado.....	4
Texto sublime.....	4
Instalación de la interfaz de línea de comandos.....	4
Instala Node.js.....	4
Instale el paquete npm globalmente.....	4
Instale el paquete npm localmente.....	4
Canales de instalación.....	5
Compilando el código TypeScript.....	5
Compilar utilizando tsconfig.json.....	5
Hola Mundo.....	5
Sintaxis basica.....	6
Declaraciones de tipo.....	6
Fundición.....	7
Las clases.....	7
TypeScript REPL en Node.js.....	8
Ejecutando TypeScript usando ts-node.....	8
Capítulo 2: ¿Por qué y cuándo usar TypeScript?.....	10
Introducción.....	10

Observaciones.....	10
Examples.....	11
La seguridad.....	11
Legibilidad.....	11
Estampación.....	12
Capítulo 3: Arrays.....	13
Examples.....	13
Encontrar objeto en matriz.....	13
Usando find ().....	13
Capítulo 4: Cómo utilizar una biblioteca javascript sin un archivo de definición de tipo.....	14
Introducción.....	14
Examples.....	14
Declarar y cualquier global.....	14
Haz un módulo que exporte un valor predeterminado.....	14
Usa un modulo ambiental.....	15
Capítulo 5: Configure el proyecto de mecanografia para compilar todos los archivos en meca... 16	16
Introducción.....	16
Examples.....	16
Configuración del archivo de configuración Typescript.....	16
Capítulo 6: Controles nulos estrictos.....	18
Examples.....	18
Controles nulos estrictos en acción.....	18
Aserciones no nulas.....	18
Capítulo 7: Decorador de clase.....	20
Parámetros.....	20
Examples.....	20
Decorador de clase basica.....	20
Generando metadatos utilizando un decorador de clase.....	20
Pasando argumentos a un decorador de clase.....	21
Capítulo 8: Depuración.....	23
Introducción.....	23

Examples.....	23
JavaScript con SourceMaps en Visual Studio Code.....	23
JavaScript con SourceMaps en WebStorm.....	23
TypeScript con ts-node en Visual Studio Code.....	24
TypeScript con ts-node en WebStorm.....	25
Capítulo 9: Enums	27
Examples.....	27
Cómo obtener todos los valores de enumeración.....	27
Enums con valores explícitos.....	27
Implementación de enumeración personalizada: se extiende para enumeraciones.....	28
Extender enumeraciones sin implementación de enumeración personalizada.....	29
Capítulo 10: Examen de la unidad.....	30
Examples.....	30
alsaciano.....	30
complemento chai-inmutable.....	30
cinta.....	31
broma.....	32
Cobertura de código.....	33
Capítulo 11: Funciones.....	36
Observaciones.....	36
Examples.....	36
Parámetros opcionales y predeterminados.....	36
Tipos de funciones.....	36
Función como parámetro.....	37
Funciones con tipos de unión.....	38
Capítulo 12: Genéricos.....	40
Sintaxis.....	40
Observaciones.....	40
Examples.....	40
Interfaces genéricas.....	40
Declarar una interfaz genérica.....	40
Interfaz genérica con múltiples parámetros de tipo.....	41

Implementando una interfaz genérica	41
Clase genérica	41
Restricciones genéricas	42
Funciones genéricas	42
Usando clases y funciones genéricas:	43
Escriba los parámetros como restricciones	43
Capítulo 13: Importando bibliotecas externas	45
Sintaxis	45
Observaciones	45
Examples	45
Importando un módulo desde npm	46
Buscando archivos de definición	46
Usando bibliotecas externas globales sin tipografías	47
Buscando archivos de definición con mecanografiado 2.x	47
Capítulo 14: Integración con herramientas de construcción	49
Observaciones	49
Examples	49
Instalar y configurar cargadores webpack +	49
Browserify	49
Instalar	49
Usando la interfaz de línea de comandos	49
Usando API	50
Gruñido	50
Instalar	50
Basic Gruntfile.js	50
Trago	50
Instalar	50
Gulpfile.js básico	50
gulpfile.js usando un tsconfig.json existente	51
Webpack	51
Instalar	51

Webpack.config.js básico	51
webpack 2.x, 3.x	51
webpack 1.x	52
MSBuild	52
NuGet	53
Capítulo 15: Interfaces	54
Introducción	54
Sintaxis	54
Observaciones	54
Interfaces vs Alias de Tipo	54
Documentación oficial de la interfaz	55
Examples	55
Añadir funciones o propiedades a una interfaz existente	55
Interfaz de clase	55
Interfaz de extensión	56
Uso de interfaces para hacer cumplir tipos	56
Interfaces genéricas	57
Declaración de parámetros genéricos en interfaces	57
Implementando interfaces genéricas	58
Usando interfaces para el polimorfismo	59
Implementación implícita y forma del objeto	60
Capítulo 16: Las clases	61
Introducción	61
Examples	61
Clase simple	61
Herencia básica	61
Constructores	62
Accesorios	63
Clases abstractas	63
Monkey parchea una función en una clase existente	64
Transpilacion	65
Fuente de TypeScript	65

Fuente de JavaScript	65
Observaciones	66
Capítulo 17: Mixins	67
Sintaxis	67
Parámetros	67
Observaciones	67
Examples	67
Ejemplo de Mixins	67
Capítulo 18: Módulos - exportación e importación	69
Examples	69
Hola módulo mundial	69
Exportación / Importación de declaraciones	69
Reexportar	70
Capítulo 19: Publicar archivos de definición de TypeScript	73
Examples	73
Incluir archivo de definición con biblioteca en npm	73
Capítulo 20: Tipo de guardias definidos por el usuario	74
Sintaxis	74
Observaciones	74
Examples	74
Usando instanceof	74
Utilizando typeof	75
Funciones de protección de tipo	75
Capítulo 21: Tipografía de ejemplos básicos	77
Observaciones	77
Examples	77
1 ejemplo básico de herencia de clase usando extended y super keyword	77
2 ejemplo de variable de clase estática: cuente la cantidad de tiempo que se invoca el mét.	77
Capítulo 22: Tipos básicos de TypeScript	79
Sintaxis	79
Examples	79

Booleano.....	79
Número.....	79
Cuerda.....	79
Formación.....	79
Enumerar.....	80
Alguna.....	80
Vacío.....	80
Tupla.....	80
Tipos en argumentos de función y valor de retorno. Número.....	81
Tipos en argumentos de función y valor de retorno. Cuerda.....	81
Tipos de cuerdas literales.....	82
Tipos de interseccion.....	85
const Enum.....	86
Capítulo 23: tsconfig.json.....	88
Sintaxis.....	88
Observaciones.....	88
Visión general.....	88
Utilizando tsconfig.json.....	88
Detalles.....	88
Esquema.....	89
Examples.....	89
Crear proyecto de TypeScript con tsconfig.json.....	89
compileOnSave.....	91
Comentarios.....	91
Configuración para menos errores de programación.....	92
preserveConstEnums.....	92
Capítulo 24: TSLint - asegurando la calidad y consistencia del código.....	94
Introducción.....	94
Examples.....	94
Configuración básica de tslint.json.....	94
Configuración para menos errores de programación.....	94
Usando un conjunto de reglas predefinido como predeterminado.....	95

Instalación y configuración.....	96
Conjuntos de Reglas TSLint.....	96
Capítulo 25: TypeScript con AngularJS.....	97
Parámetros.....	97
Observaciones.....	97
Examples.....	97
Directiva.....	97
Ejemplo simple.....	98
Componente.....	99
Capítulo 26: TypeScript con SystemJS.....	101
Examples.....	101
Hola mundo en el navegador con SystemJS.....	101
Capítulo 27: Typescript-installation-typescript-and-running-the-typescript-compiler-tsc.....	104
Introducción.....	104
Examples.....	104
Pasos.....	104
Instalar Typescript y ejecutar el compilador de typScript.....	104
Capítulo 28: Usando Typescript con React (JS & native).....	107
Examples.....	107
Componente ReactJS escrito en Typescript.....	107
Mecanografiado y reaccionar y paquete web.....	108
Capítulo 29: Usando Typescript con RequireJS.....	110
Introducción.....	110
Examples.....	110
Ejemplo de HTML usando requireJS CDN para incluir un archivo TypeScript ya compilado.....	110
Ejemplo de tsconfig.json para compilar para ver la carpeta usando el estilo de importación.....	110
Capítulo 30: Usando TypeScript con webpack.....	111
Examples.....	111
webpack.config.js.....	111
Creditos.....	113

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [typescript](#)

It is an unofficial and free TypeScript ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official TypeScript.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con TypeScript

Observaciones

TypeScript pretende ser un superconjunto de JavaScript que se traslada a JavaScript. Al generar código compatible con ECMAScript, TypeScript puede introducir nuevas características de lenguaje al tiempo que conserva la compatibilidad con los motores de JavaScript existentes. ES3, ES5 y ES6 son actualmente objetivos compatibles.

Los tipos opcionales son una característica principal. Los tipos permiten la verificación estática con el objetivo de encontrar errores antes y pueden mejorar las herramientas con características como la refactorización de códigos.

TypeScript es un lenguaje de programación de código abierto y multiplataforma desarrollado por Microsoft. El [código fuente está disponible en GitHub](#).

Versiones

Versión	Fecha de lanzamiento
2.4.1	2017-06-27
2.3.2	2017-04-28
2.3.1	2017-04-25
2.3.0 beta	2017-04-04
2.2.2	2017-03-13
2.2	2017-02-17
2.1.6	2017-02-07
2.2 beta	2017-02-02
2.1.5	2017-01-05
2.1.4	2016-12-05
2.0.8	2016-11-08
2.0.7	2016-11-03
2.0.6	2016-10-23
2.0.5	2016-09-22

Versión	Fecha de lanzamiento
2.0 Beta	2016-07-08
1.8.10	2016-04-09
1.8.9	2016-03-16
1.8.5	2016-03-02
1.8.2	2016-02-17
1.7.5	2015-12-14
1.7	2015-11-20
1.6	2015-09-11
1.5.4	2015-07-15
1.5	2015-07-15
1.4	2015-01-13
1.3	2014-10-28
1.1.0.1	2014-09-23

Examples

Instalación y configuración

Fondo

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

IDEs

Estudio visual

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

Código de Visual Studio

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

Tormenta web

- [WebStorm 2016.2](#) viene con TypeScript y un compilador incorporado. [Webstorm no es gratis]

IntelliJ IDEA

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

Atom & atom-mecanografiado

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

Texto sublime

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

Instalación de la interfaz de línea de comandos

Instala [Node.js](#)

Instale el paquete npm globalmente

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

```
npm install -g typescript
```

O

Instale el paquete npm localmente

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

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

Canales de instalación

Se puede instalar desde:

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

Compilando el código TypeScript

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

```
tsc my-code.ts
```

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

Compilar utilizando tsconfig.json

También puede proporcionar opciones de compilación que viajan con su código a través de un archivo `tsconfig.json`. Para iniciar un nuevo proyecto mecanografiado, `cd` en el directorio raíz de su proyecto en una ventana de terminal y ejecute `tsc --init`. Este comando generará un archivo `tsconfig.json` con opciones de configuración mínimas, similares a las siguientes.

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

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

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 de `greet`. Podemos construir una instancia de la clase usando la `new` palabra clave y pasar una cadena que queremos que el método de `greet` salga a la consola. La instancia de nuestra clase `Greeter` se almacena en la variable `greeter` que luego llamamos el método `greet`.

Sintaxis basica

TypeScript es un superconjunto escrito de JavaScript, lo que significa que todo el código JavaScript es un código TypeScript válido. TypeScript agrega muchas características nuevas además de eso.

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

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

Declaraciones de tipo

Puede agregar declaraciones de tipo a variables, parámetros de función y tipos de retorno de función. El tipo se escribe después de dos puntos que siguen al nombre de la variable, como este: `var num: number = 5;` El compilador luego verificará los tipos (cuando sea posible) durante la compilación y los errores de tipo de informe.

```
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 números de punto flotante)
- `string`
- `boolean`
- `Array` Puede especificar los tipos de elementos de una matriz. Hay dos formas equivalentes de definir los tipos de matriz: `Array<T>` y `T[]`. Por ejemplo:
 - `number[]` - matriz de números
 - `Array<string>` - array of strings
- `Tuplas`. Las tuplas tienen un número fijo de elementos con tipos específicos.
 - `[boolean, string]` - tupla donde el primer elemento es booleano y el segundo es una cadena.

- `[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}` - un diccionario de números indexados por cadena
- `enum` - `{ Red = 0, Blue, Green }` - enumeración asignada a números
- **Función.** Usted especifica los tipos para los parámetros y el valor de retorno:
 - `(param: number) => string` - función que toma un parámetro numérico que devuelve una cadena
 - `() => number` - función sin parámetros que devuelven un número.
 - `(a: string, b?: boolean) => void` : función que toma una cadena y, opcionalmente, un booleano sin valor de retorno.
- `any` - Permite cualquier tipo. No se comprueban las expresiones que involucren a `any`.
- `void` - representa "nada", se puede utilizar como valor de retorno de una función. Solo `null` y `undefined` son parte del tipo `void`.
- `never`
 - `let foo: never;` - Como el tipo de variables en las guardas de tipo que nunca son ciertas.
 - `function error(message: string): never { throw new Error(message); }` - Como el tipo de retorno de funciones que nunca regresan.
- `null` : escriba el valor `null`. `null` es implícitamente parte de cada tipo, a menos que las comprobaciones nulas estrictas estén habilitadas.

Fundición

Puede realizar una conversión explícita entre paréntesis angulares, por ejemplo:

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

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

Otra forma de convertir en Typescript es usando la palabra clave `as`:

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

Desde Typescript 1.6, el valor predeterminado es usar la palabra clave `as`, porque usar `<>` es ambiguo en los archivos `.jsx`. Esto se menciona en la [documentación oficial de Typescript](#).

Las clases

Las clases se pueden definir y utilizar en el código de TypeScript. Para obtener más información sobre las clases, consulte la [página de documentación de Clases](#).

TypeScript REPL en Node.js

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

Instálalo globalmente con

```
npm install -g tsun
```

y ejecútelo en su terminal o indicador de comando 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
```

Ejecutando TypeScript usando ts-node

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

Instale ts-node globalmente usando

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

ts-node no incluye el compilador de escritura de tipos, por lo que es posible que deba instalarlo.

```
npm install -g typescript
```

Ejecutando 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
```

Ejecutando 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 use el comando `.exit` o presione `CTRL+C` dos veces.

Lea Empezando con TypeScript en línea:

<https://riptutorial.com/es/typescript/topic/764/empezando-con-typescript>

Capítulo 2: ¿Por qué y cuándo usar TypeScript?

Introducción

Si encuentra que los argumentos para los sistemas de tipos son persuasivos en general, entonces estará satisfecho con TypeScript.

Aporta muchas de las ventajas del sistema de tipos (seguridad, legibilidad, herramientas mejoradas) al ecosistema de JavaScript. También sufre de algunos de los inconvenientes de los sistemas tipográficos (complejidad agregada e incompleta).

Observaciones

Los méritos de los lenguajes mecanografiados frente a los no tipificados han sido debatidos durante décadas. Los argumentos para los tipos estáticos incluyen:

1. Seguridad: los sistemas de tipo permiten detectar muchos errores antes de tiempo, sin ejecutar el código. TypeScript puede [configurarse para permitir menos errores de programación](#)
2. Legibilidad: los tipos explícitos hacen que el código sea más fácil de entender para los humanos. Como [escribió](#) Fred Brooks, "Muéstrame tus diagramas de flujo y oculta tus tablas, y seguiré desconcertado. Muéstrame tus tablas, y generalmente no necesitaré tus diagramas de flujo; serán obvios".
3. Herramientas: los sistemas tipográficos hacen que el código sea más fácil de entender para las computadoras. Esto permite que herramientas como IDE y linters sean más poderosas.
4. Rendimiento: los sistemas de tipo hacen que el código se ejecute más rápido al reducir la necesidad de verificar el tipo de tiempo de ejecución.

Debido a que [la salida de TypeScript es independiente de sus tipos](#), TypeScript no tiene impacto en el rendimiento. El argumento para usar TypeScript se basa en las otras tres ventajas.

Los argumentos contra los sistemas tipográficos incluyen:

1. Complejidad agregada: los sistemas de tipos pueden ser más complejos que el tiempo de ejecución de lenguaje que describen. Las funciones de orden superior pueden ser fáciles de implementar correctamente pero [difíciles de escribir](#). Tratar con las definiciones de tipo crea barreras adicionales para el uso de bibliotecas externas.
2. Verbo añadido: las anotaciones de tipo pueden agregar repetitivo a su código, haciendo que la lógica subyacente sea más difícil de seguir.
3. Una iteración más lenta: al introducir un paso de compilación, TypeScript ralentiza el ciclo de edición / guardar / recargar.
4. Incompleto: un sistema de tipo no puede ser completo y completo. Hay programas correctos que TypeScript no permite. Y los programas que acepta TypeScript pueden contener

errores. Un sistema de tipo no alivia la necesidad de pruebas. Si usa TypeScript, es posible que tenga que esperar más para usar las nuevas funciones de lenguaje ECMAScript.

TypeScript ofrece algunas formas de abordar todos estos problemas:

1. Mayor complejidad. Si escribir parte de un programa es demasiado difícil, TypeScript se puede deshabilitar en gran medida utilizando `any` tipo opaco. Lo mismo ocurre con los módulos externos.
2. Verbo añadido. Esto se puede abordar parcialmente a través de alias de tipo y la capacidad de TypeScript para inferir tipos. Escribir un código claro es tanto un arte como una ciencia: elimine demasiadas anotaciones de tipo y es posible que el código ya no sea claro para los lectores humanos.
3. Una iteración más lenta: un paso de compilación es relativamente común en el desarrollo moderno de JS y TypeScript ya se [integra con la mayoría de las herramientas de compilación](#) . ¡Y si TypeScript detecta un error antes, puede ahorrarle un ciclo completo de iteración!
4. Incompletitud. Si bien este problema no se puede resolver completamente, TypeScript ha podido capturar patrones de JavaScript cada vez más expresivos a lo largo del tiempo. Los ejemplos recientes incluyen la adición de [tipos mapeados en TypeScript 2.1](#) y [mixins en 2.2](#).

Los argumentos a favor y en contra de los sistemas tipográficos en general se aplican igualmente a TypeScript. El uso de TypeScript aumenta la sobrecarga para iniciar un nuevo proyecto. Pero con el tiempo, a medida que el proyecto aumenta de tamaño y gana más contribuyentes, la esperanza es que las ventajas de usarlo (seguridad, legibilidad, herramientas) se vuelvan más fuertes y superen las desventajas. Esto se refleja en el lema de TypeScript: "JavaScript que escala".

Examples

La seguridad

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

```
function double(x: number): number {
  return 2 * x;
}
double('2');
//      ~~~ Argument of type '"2"' is not assignable to parameter of type 'number'.
```

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 olvidará si `String.prototype.slice` toma `(start, stop)` o `(start, length)` otra vez!

Estampación

TypeScript permite a los editores realizar refactoros automáticos que conocen las reglas de los idiomas.

```
let foo = '123';

{
  const foo = (x: number) => {
    return 2 * x;
  }

  foo(2);
}
```

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

Lea ¿Por qué y cuándo usar TypeScript? en línea: <https://riptutorial.com/es/typescript/topic/9073/-por-que-y-cuando-usar-typescript->

Capítulo 3: Arrays

Examples

Encontrar objeto en matriz

Usando 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 }

/* OR */

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

Lea Arrays en línea: <https://riptutorial.com/es/typescript/topic/9562/arrays>

Capítulo 4: Cómo utilizar una biblioteca javascript sin un archivo de definición de tipo

Introducción

Si bien algunas bibliotecas de JavaScript existentes tienen [archivos de definición de tipo](#) , hay muchas que no lo hacen.

TypeScript ofrece un par de patrones para manejar las declaraciones faltantes.

Examples

Declarar y cualquier global

A veces es más fácil simplemente declarar un global de tipo `any` , sobre todo en proyectos simples.

Si jQuery no tenía declaraciones de tipo (las [tiene](#)), podría poner

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

Ahora cualquier uso de `$` será escrito `any` .

Haz un módulo que exporte un valor predeterminado.

Para proyectos más complicados, o en los casos en los que tiene la intención de escribir gradualmente una dependencia, puede ser más limpio crear un módulo.

Usando JQuery (aunque [tiene tipografías disponibles](#)) como ejemplo:

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

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

```
// some other .ts file
import $ from "jquery";
```

Después de esta importación, `$` se escribirá como `any` .

Si la biblioteca tiene varias variables de nivel superior, exporte e importe por nombre en su lugar:

```
// place in jquery.d.ts
declare module "jquery" {
```

```
let $: any;
let jQuery: any;

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

A continuación, puede importar y usar ambos nombres:

```
// some other .ts file
import { $, jQuery } from "jquery";

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

Usa un modulo ambiental

Si solo 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 ambiente.

```
// in a declarations file (like declarations.d.ts)
declare module "jquery";    // note that there are no defined exports
```

A continuación, puede importar desde el módulo ambiente.

```
// some other .ts file
import { $, jQuery } from "jquery";
```

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

Lea [Cómo utilizar una biblioteca javascript sin un archivo de definición de tipo en línea](https://riptutorial.com/es/typescript/topic/8249/como-utilizar-una-biblioteca-javascript-sin-un-archivo-de-definicion-de-tipo):
<https://riptutorial.com/es/typescript/topic/8249/como-utilizar-una-biblioteca-javascript-sin-un-archivo-de-definicion-de-tipo>

Capítulo 5: Configure el proyecto de mecanografía para compilar todos los archivos en mecanografía

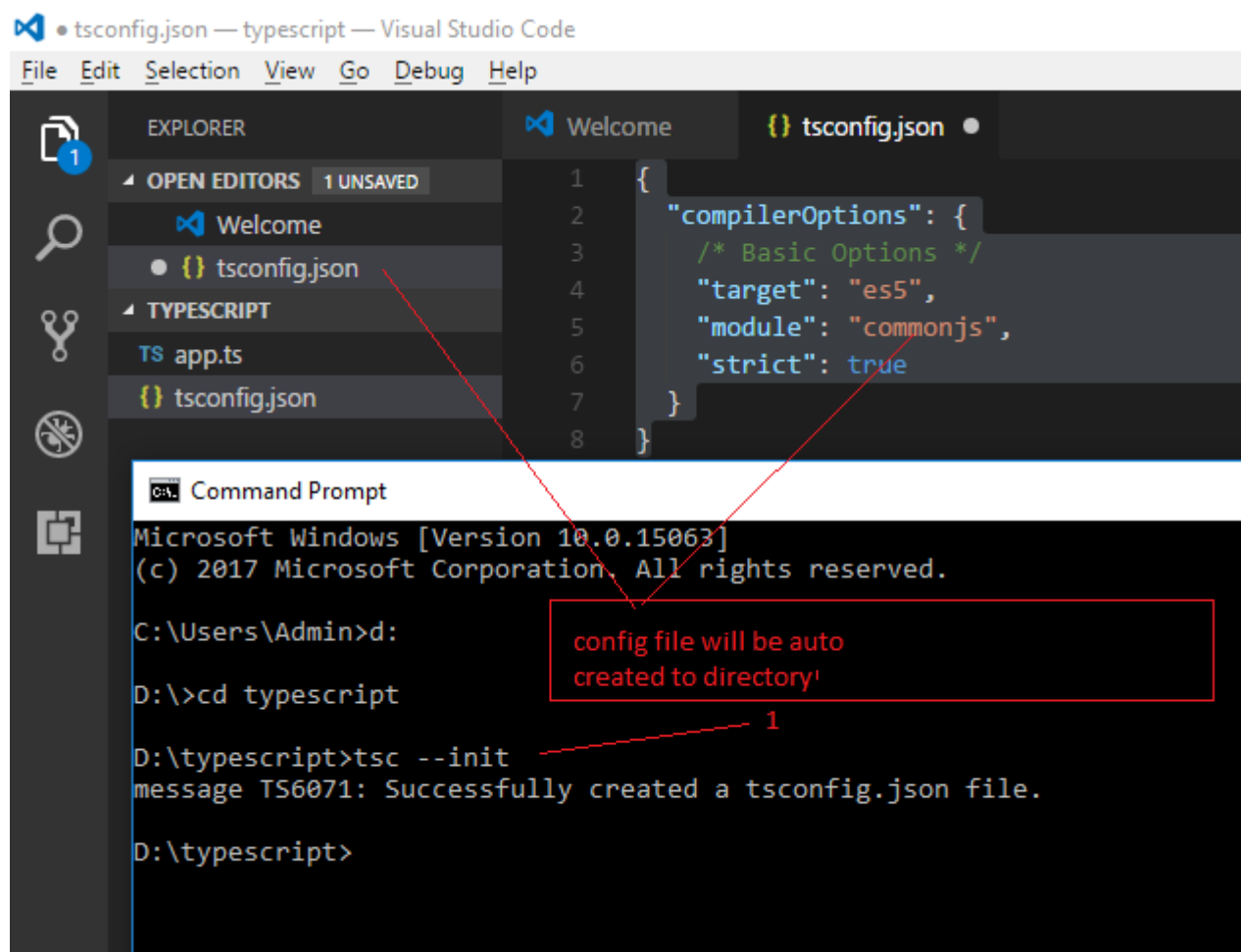
Introducción

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

Examples

Configuración del archivo de configuración Typescript

- Ingrese el comando `"tsc --init"` y **presione** enter.
- Antes de eso, debemos compilar el archivo ts con el comando `"tsc app.ts"`, ahora todo está definido en el archivo de configuración siguiente automáticamente.



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

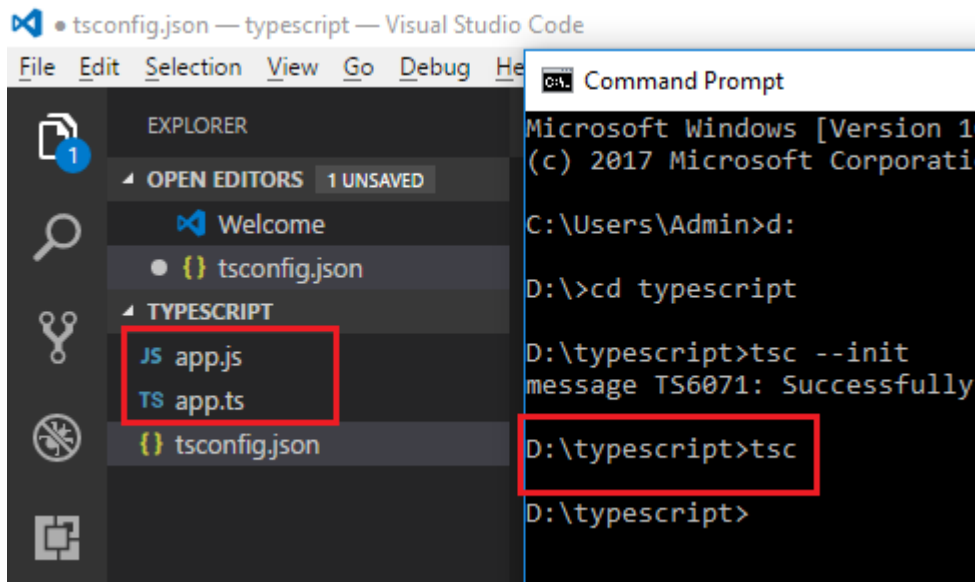
```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",
    "module": "commonjs",
    "strict": true
  }
}
```

Below the editor, a Command Prompt window is open, showing the following commands and output:

```
C:\Users\Admin>d:
D:\>cd typescript
D:\typescript>tsc --init
message TS6071: Successfully created a tsconfig.json file.
D:\typescript>
```

A red box highlights the output message in the Command Prompt, with a red arrow pointing to the `tsconfig.json` file in the Explorer sidebar. A red line also points from the `tsconfig.json` file in the Explorer sidebar to the `tsconfig.json` file in the editor.

- Ahora, puede compilar todos los tipos de escritura con el comando `"tsc"`. creará automáticamente el archivo `".js"` de su archivo mecanografiado.



- Si creará otra escritura de tipo y pulsará el comando "tsc" en el símbolo del sistema o el archivo de terminal javascript se creará automáticamente para el archivo de escritura.

Gracias,

Lea Configure el proyecto de mecanografía para compilar todos los archivos en mecanografía en línea: <https://riptutorial.com/es/typescript/topic/10537/configure-el-proyecto-de-mecanografia-para-compilar-todos-los-archivos-en-mecanografia>

Capítulo 6: Controles nulos estrictos

Examples

Controles nulos estrictos en acción.

De forma predeterminada, todos los tipos en TypeScript permiten `null` :

```
function getId(x: Element) {  
    return x.id;  
}  
getId(null); // TypeScript does not complain, but this is a runtime error.
```

TypeScript 2.0 agrega soporte para controles nulos estrictos. Si establece `--strictNullChecks` cuando ejecuta `tsc` (o establece este indicador en su `tsconfig.json`), entonces los tipos ya no permiten `null` :

```
function getId(x: Element) {  
    return x.id;  
}  
getId(null); // error: Argument of type 'null' is not assignable to parameter of type  
'Element'.
```

Debe permitir valores `null` explícitamente:

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

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

```
function getId(x: Element|null) {  
    if (x) {  
        return x.id; // In this branch, x's type is Element  
    } else {  
        return null; // In this branch, x's type is null.  
    }  
}  
getId(null);
```

Aserciones no nulas

El operador afirmación no nulo, `!` , le permite afirmar que una expresión no es `null` o `undefined` cuando el compilador de TypeScript no puede inferir eso 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);

    // Even though we know `node.next` is defined because we just called `addNext`,
    // TypeScript isn't able to infer this in the line of code below:
    // node.next.data = value;

    // So, we can use the non-null assertion operator, !,
    // to assert that node.next isn't undefined and silence the compiler warning
    node.next!.data = value;
}
```

Lea Controles nulos estrictos en línea: <https://riptutorial.com/es/typescript/topic/1727/controles-nulos-estrictos>

Capítulo 7: Decorador de clase

Parámetros

Parámetro	Detalles
objetivo	La clase siendo decorada

Examples

Decorador de clase basica

Un decorador de clase es solo una función que toma a la clase como su único argumento y la devuelve después de hacer algo con ella:

```
function log<T>(target: T) {  
  
    // Do something with target  
    console.log(target);  
  
    // Return target  
    return target;  
  
}
```

Entonces 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;  
    }  
}
```

Generando metadatos utilizando un decorador de clase.

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

```
function addMetadata(target: any) {  
  
    // Add some metadata  
    target.__customMetadata = {  
        someKey: "someValue"  
    };  
  
}
```

```

    // Return target
    return target;
}

```

Entonces 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 la clase no se declara cuando creamos instancias de la clase. Esto significa que los metadatos se comparten en 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));

```

Pasando 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) {

        // Add metadata
        target.__customMetadata = metadata;

        // Return 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.

Entonces podemos invocar al 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 usar la siguiente función para acceder a los metadatos generados:

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

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

Si todo salió bien, la consola debería mostrar:

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

Lea Decorador de clase en línea: <https://riptutorial.com/es/typescript/topic/4592/decorador-de-clase>

Capítulo 8: Depuración

Introducción

Hay dos formas de ejecutar y depurar TypeScript:

Transpile a JavaScript , ejecútelo en un nodo y use mapeos para enlazar de nuevo a los archivos de origen de TypeScript

o

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

Este artículo describe las dos formas de usar [Visual Studio Code](#) y [WebStorm](#) . Todos los ejemplos suponen que su archivo principal es *index.ts* .

Examples

JavaScript con SourceMaps en Visual Studio Code

En el conjunto `tsconfig.json`

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

para generar asignaciones junto con archivos js desde las fuentes de TypeScript usando el comando `tsc` .

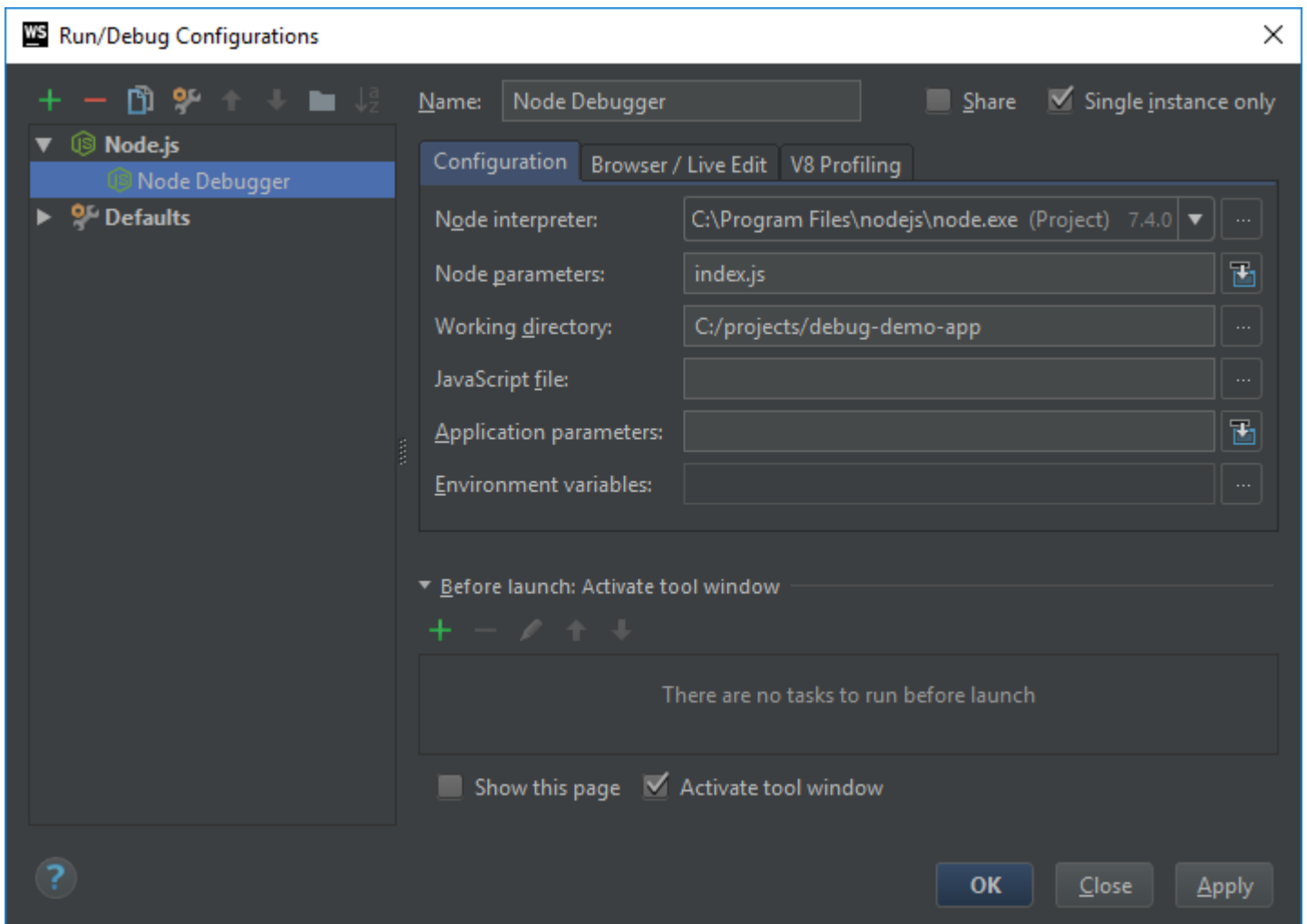
El archivo [launch.json](#) :

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceRoot}\\index.js",
      "cwd": "${workspaceRoot}",
      "outFiles": [],
      "sourceMaps": true
    }
  ]
}
```

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

JavaScript con SourceMaps en WebStorm

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



TypeScript con ts-node en Visual Studio Code

Agregue ts-node a su proyecto de TypeScript:

```
npm i ts-node
```

Agrega 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 utilizar el tipo de *nodo 2* y empezar NPM ejecutar el `start:debug` de script:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node2",
      "request": "launch",
      "name": "Launch Program",
      "runtimeExecutable": "npm",
      "windows": {
```

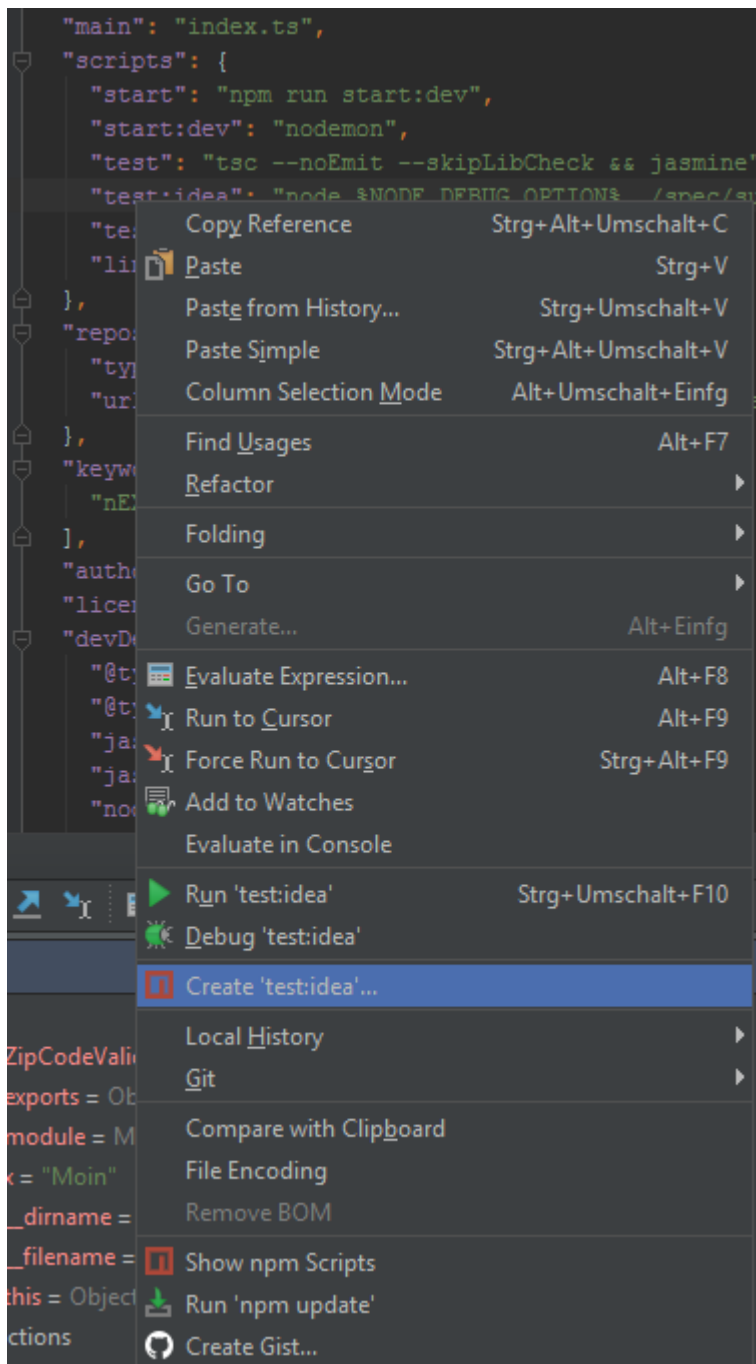
```
        "runtimeExecutable": "npm.cmd"
      },
      "runtimeArgs": [
        "run-script",
        "start:debug"
      ],
      "cwd": "${workspaceRoot}/server",
      "outFiles": [],
      "port": 5858,
      "sourceMaps": true
    }
  ]
}
```

TypeScript con ts-node en WebStorm

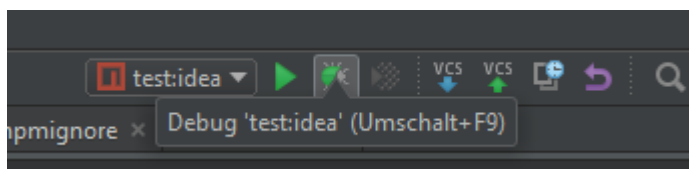
Agregue este script a su `package.json` :

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

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



Inicie el depurador usando esta configuración:



Lea Depuración en línea: <https://riptutorial.com/es/typescript/topic/9131/depuracion>

Capítulo 9: Enums

Examples

Cómo obtener todos los valores de enumeración

```
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
```

Enums con valores explícitos.

Por defecto, todos los valores de `enum` se resuelven en números. Digamos que si tienes algo como

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

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

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

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

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

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

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

```
}
```

Puedes proporcionar explícitamente valores numéricos usando el mismo método

```
enum MyType {  
    Value = 3,  
    ValueEx = 30,  
    ValueEx2 = 300  
}
```

Los tipos más sofisticados también funcionan, ya que las no constantes 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 = {}));
```

Implementación de enumeración personalizada: se extiende para enumeraciones

A veces es necesario implementar Enum por su cuenta. Por ejemplo, no hay una forma clara de extender otras enumeraciones. 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));
// this works but perhaps your TSLint would complain
// attention! does not work with ===
// use .is() instead
console.log(TestEnum.value1 == <any>'value1');

```

Extender enumeraciones sin implementación de enumeración personalizada

```

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

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

// we need this type for TypeScript to resolve the types correctly
type TestEnumType = SourceEnum | AdditionToSourceEnum;
// and we need this value "instance" to use values
let TestEnum = Object.assign({}, SourceEnum, AdditionToSourceEnum);
// also works fine the TypeScript 2 feature
// 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));

```

Lea Enums en línea: <https://riptutorial.com/es/typescript/topic/4954/enums>

Capítulo 10: Examen de la unidad

Examples

alsaciano

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

Para usarlo, instálalo desde `npm`:

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

A continuación, configure 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, vea el [repositorio de GitHub de alsatian](#).

complemento chai-inmutable

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

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

2. Instala tipos para mocha y chai.

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

3. Escribir archivo de prueba simple:

```
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. Ejecutarlo en la consola:

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

cinta

la [cinta](#) es un marco de prueba de JavaScript minimalista, genera un [marcado compatible con TAP](#).

Para instalar la `tape` usando el comando `npm run`.

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

Para usar la `tape` con Typescript necesita instalar `ts-node` como paquete global, para ejecutar este comando

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

Ahora estás listo para escribir tu 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 test run

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

En salida deberías 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 de TypeScript.

Ejecutar múltiples archivos de prueba

Puede ejecutar varios archivos de prueba a la vez utilizando comodines de ruta. Para ejecutar todas las pruebas de Typescript en el comando de ejecución del directorio de `tests`

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

broma

[jest](#) es un marco de prueba de JavaScript sin dolor por Facebook, con [ts-jest](#) se puede utilizar para probar el código TypeScript.

Para instalar jest usando el comando npm run

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

Para facilitar el uso, instale `jest` como paquete global.

```
npm install -g jest
```

Para hacer que `jest` funcione con TypeScript, debe agregar la configuración a `package.json`

```
//package.json
{
  ...
  "jest": {
    "transform": {
      "^.+\\.tsx?$": "ts-jest"
    },
    "testRegex": "(/__tests__/.*\\.?(test|spec))\\.?(ts|tsx|js)$",
  },
}
```

```
    "moduleFileExtensions": ["ts", "tsx", "js"]
  }
}
```

Ahora la `jest` está lista. Supongamos que tenemos una muestra de fizz buz para probar

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

Ejemplo de prueba podría verse como

```
//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 ejecución de prueba

```
jest
```

En salida deberías 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 de código

`jest` soporta la generación de informes de cobertura de código.

Para usar la cobertura de código con TypeScript, debe agregar otra línea de configuración a `package.json`.

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

Ejecutar pruebas con generación de informes de cobertura ejecutados.

```
jest --coverage
```

Si se usa con nuestro zumbido de muestra, debería 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 creó una `coverage` carpeta que contiene un informe 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

File ▲		Statements ▾	
fizzBuzz.ts		92.31%	12/1

Lea Examen de la unidad en línea: <https://riptutorial.com/es/typescript/topic/7456/examen-de-la-unidad>

Capítulo 11: Funciones

Observaciones

Enlace de documentación mecanografiado para [Funciones](#)

Examples

Parámetros opcionales y predeterminados

Parámetros opcionales

En TypeScript, se supone que cada parámetro es requerido por la función. Usted puede agregar un `?` al final de un nombre de parámetro para configurarlo como opcional.

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

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

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

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

Parámetros predeterminados

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

Por ejemplo, "Smith" es el valor predeterminado para el 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'
```

Tipos de funciones

Funciones nombradas

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

Funciones anonimas

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

Funciones Lambda / flecha

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

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 facilitar la lectura, podemos definir una interfaz que describa al constructor:

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

O con parámetros:

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

Incluso con los genéricos:

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

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

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

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

function foo(constructorWithParamsFunc: { (num: number): void }) {
    new constructorWithParamsFunc(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 los genéricos:

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

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

Funciones con tipos de unión

Una función de TypeScript puede admitir 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 tipo único que es una unión de los otros tipos, por lo que su función debe ser capaz de manejar parámetros de cualquier tipo que estén en la unión.

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

Lea Funciones en línea: <https://riptutorial.com/es/typescript/topic/1841/funciones>

Capítulo 12: Genéricos

Sintaxis

- Los tipos genéricos declarados dentro de los corchetes triangulares: `<T>`
- La restricción de los tipos genéricos se realiza con la palabra clave `extends`: `<T extends Car>`

Observaciones

Los parámetros genéricos no están disponibles en tiempo de ejecución, son solo para el tiempo de compilación. Esto significa que no puedes hacer algo como esto:

```
class Executor<T, U> {
    public execute(executable: T): void {
        if (T instanceof Executable1) {    // Compilation error
            ...
        } else if (U instanceof Executable2){    // Compilation error
            ...
        }
    }
}
```

Sin embargo, la información de la clase aún se conserva, por lo que aún puede probar el tipo de variable como siempre ha podido:

```
class Executor<T, U> {
    public execute(executable: T): void {
        if (executable instanceof Executable1) {
            ...
        } else if (executable instanceof Executable2){
            ...
        } // But in this method, since there is no parameter of type `U` it is non-sensical to
        ask about U's "type"
    }
}
```

Examples

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);
```

Implementando una interfaz genérica

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

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

Implementarlo con 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);
    }
}
```

Implementarlo con 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);
    }
}
```

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

```

Restricciones genéricas

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;
}

```

Funciones genéricas

En interfaces:

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

En clases:

```
class Runner implements IRunner {

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

}
```

Funciones simples:

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

Usando 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);

// Specify the generic types:
runSafe<Runnable<string>, string>(stringRunnable);

// Let typescript figure the generic types by himself:
runSafe(stringRunnable);
```

Escriba los parámetros como restricciones

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

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

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

Lea Genéricos en línea: <https://riptutorial.com/es/typescript/topic/2132/genericos>

Capítulo 13: Importando bibliotecas externas

Sintaxis

- `import {component} from 'libName';` // Will import the class "component"
- `import {component as c} from 'libName';` // Will import the class "component" into a "c" object
- `import component from 'libname';` // Will import the default export from libName
- `import * as lib from 'libName';` // Will import everything from libName into a "lib" object
- `import lib = require('libName');` // Will import everything from libName into a "lib" object
- `const lib: any = require('libName');` // Will import everything from libName into a "lib" object
- `import 'libName';` // Will import libName module for its side effects only

Observaciones

Podría parecer que la sintaxis

```
import * as lib from 'libName';
```

y

```
import lib = require('libName');
```

son lo mismo, pero no lo son!

Consideremos que queremos importar una clase **Persona** exportada con sintaxis `export =` específica de TypeScript:

```
class Person {  
  ...  
}  
export = Person;
```

En este caso, no es posible importarlo con la sintaxis es6 (obtendríamos un error en el momento de la compilación), se debe utilizar la sintaxis `import =` específica de TypeScript.

```
import * as Person from 'Person'; //compile error  
import Person = require('Person'); //OK
```

Lo contrario es cierto: los módulos clásicos se pueden importar con la segunda sintaxis, por lo que, de alguna manera, la última sintaxis es más poderosa ya que es capaz de importar todas las exportaciones.

Para más información ver la [documentación oficial](#) .

Examples

Importando un módulo desde npm

Si tiene un archivo de definición de tipo (d.ts) para el módulo, puede usar una declaración de `import`.

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

Si no tiene un archivo de definición para el módulo, TypeScript generará un error en la compilación porque no puede encontrar el módulo que está intentando importar.

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

```
// The _ variable is of type any, so TypeScript will not perform any type checking.
const _: any = require('lodash');
```

A partir de TypeScript 2.0, también puede usar una *declaración abreviada del módulo ambiental* para indicar a TypeScript que existe un módulo cuando no tiene un archivo de definición de tipo para el módulo. Sin embargo, TypeScript no podrá proporcionar ninguna comprobación de tipo significativa en este caso.

```
declare module "lodash";

// you can now import from lodash in any way you wish:
import { flatten } from "lodash";
import * as _ from "lodash";
```

A partir de TypeScript 2.1, las reglas se han relajado aún más. Ahora, mientras exista un módulo en su directorio `node_modules`, TypeScript le permitirá importarlo, incluso sin ninguna declaración de módulo en ningún lugar. (Tenga en cuenta que si utiliza la opción del compilador `--noImplicitAny`, lo siguiente seguirá generando una advertencia).

```
// Will work if `node_modules/someModule/index.js` exists, or if
`node_modules/someModule/package.json` has a valid "main" entry point
import { foo } from "someModule";
```

Buscando archivos de definición

para mecanografía 2.x:

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

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

pero en el caso de que desee utilizar tipos de otros repositorios, entonces puede usarlos de forma antigua:

para mecanografía 1.x:

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

```
npm install -global typings
```

Ahora tenemos acceso a las tipografías cli.

1. 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			

2. Luego decide de qué fuente deberías instalar. Utilizo dt, que significa [DefinitelyTyped](#), un repositorio de GitHub en el que la comunidad puede editar las tipografías, normalmente también es la más reciente.
3. Instala los archivos de mecanografía.

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

Vamos a romper el último comando. Estamos instalando la versión DefinitelyTyped de lodash como un archivo de mecanografía global en nuestro proyecto y guardándolo como una dependencia en `typings.json` . Ahora, dondequiera que importemos lodash, typScript cargará el archivo de tipificaciones de lodash.

4. Si queremos instalar las tipificaciones que se usarán solo para el entorno de desarrollo, podemos proporcionar el `--save-dev` :

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

Usando bibliotecas externas globales sin tipografías

Aunque los módulos son ideales, si una biblioteca global hace referencia a la biblioteca que está utilizando (como `$` o `_`), ya que se cargó con una etiqueta de `script` , puede crear una declaración de ambiente para referirse a ella:

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

Buscando archivos de definición con mecanografiado 2.x

Con las versiones 2.x de mecanografiado, las mecanografías ahora están disponibles en el [repositorio npm @types](#) . Estos son resueltos automáticamente por el compilador de

mecanografía y son mucho más fáciles de usar.

Para instalar una definición de tipo, simplemente instálala como una dependencia de desarrollo en tus proyectos package.json

p.ej

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

después de la instalación simplemente utiliza el módulo como antes

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

Lea [Importando bibliotecas externas en línea](https://riptutorial.com/es/typescript/topic/1542/importando-bibliotecas-externas):

<https://riptutorial.com/es/typescript/topic/1542/importando-bibliotecas-externas>

Capítulo 14: Integración con herramientas de construcción

Observaciones

Para obtener más información, puede ir a la página web oficial [con integración de herramientas de compilación](#)

Examples

Instalar y configurar cargadores webpack +

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/
    }],
  }
};
```

Browserify

Instalar

```
npm install tsify
```

Usando la interfaz de línea de comandos

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

Usando API

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

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

Más detalles: [smrq / tsify](#)

Gruñido

Instalar

```
npm install grunt-ts
```

Basic Gruntfile.js

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

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

Trago

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 // You can add and overwrite parameters here
});

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

Más detalles: [ivogabe / gulp-typescript](#)

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: [
      {
        // Set up ts-loader for .ts/.tsx files and exclude any imports from
node_modules.
        test: /\.tsx?$/,
        loaders: ['ts-loader'],
        exclude: /node_modules/
      }
    ]
  },
  entry: [
    // Set index.tsx as application entry point.
    './index.tsx'
  ],
  output: {
    filename: "bundle.js"
  }
};

```

webpack 1.x

```

module.exports = {
  entry: "./src/index.tsx",
  output: {
    filename: "bundle.js"
  },
  resolve: {
    // Add '.ts' and '.tsx' as a resolvable extension.
    extensions: [".", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
  },
  module: {
    loaders: [
      // all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'
      { test: /\.ts(x)?$/, loader: "ts-loader", exclude: /node_modules/ }
    ]
  }
}

```

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

Alternativas:

- [awesome-typescript-loader](#)

MSBuild

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

```

<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Include default props at the bottom -->
  <Import

```

```

Project="$ (MSBuildExtensionsPath32) \Microsoft\VisualStudio\v$(VisualStudioVersion) \TypeScript\Microsoft

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

<!-- TypeScript configurations go here -->
<PropertyGroup Condition="'$(Configuration)' == 'Debug'">
  <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
  <TypeScriptSourceMap>true</TypeScriptSourceMap>
</PropertyGroup>
<PropertyGroup Condition="'$(Configuration)' == 'Release'">
  <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
  <TypeScriptSourceMap>false</TypeScriptSourceMap>
</PropertyGroup>

<!-- Include default targets at the bottom -->
<Import

Project="$ (MSBuildExtensionsPath32) \Microsoft\VisualStudio\v$(VisualStudioVersion) \TypeScript\Microsoft

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

```

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

NuGet

- Clic derecho -> Administrar paquetes de NuGet
- **Buscar** `Microsoft.TypeScript.MSBuild`
- **Hit** `Install`
- Cuando se complete la instalación, reconstruir!

Se pueden encontrar más detalles en el [cuadro de diálogo Administrador de paquetes](#) y sobre el [uso de compilaciones nocturnas con NuGet](#)

Lea [Integración con herramientas de construcción en línea](#):

<https://riptutorial.com/es/typescript/topic/2860/integracion-con-herramientas-de-construccion>

Capítulo 15: Interfaces

Introducción

Una interfaz especifica una lista de campos y funciones que pueden esperarse en 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.

El principal beneficio de usar interfaces, es que permite usar objetos de diferentes tipos de manera polimórfica. Esto se debe a que cualquier clase que implemente la interfaz tiene al menos esos campos y funciones.

Sintaxis

- `interfaz InterfaceName {`
- `ParameterName: ParameterType;`
- `optionalParameterName?: parametersType;`
- `}`

Observaciones

Interfaces vs Alias de Tipo

Las interfaces son buenas para especificar la forma de un objeto, por ejemplo, para un objeto de persona que podría especificar

```
interface person {  
    id?: number;  
    name: string;  
    age: number;  
}
```

Sin embargo, ¿qué sucede si quiere representar, por ejemplo, la forma en que una persona se almacena en una base de datos SQL? Al ver que cada entrada de base de datos consta de una fila de forma `[string, string, number]` (por lo tanto, una matriz de cadenas o números), no hay forma de que pueda representar esto como una forma de objeto, porque la fila no tiene *propiedades*. como tal, es sólo una matriz.

Esta es una ocasión donde los tipos son útiles. En lugar de especificar en cada función que **acepte una** `function processRow(row: [string, string, number])` **parámetro de fila** `function processRow(row: [string, string, number])`, puede crear un alias de tipo separado para una fila y luego usarlo en cada función:

```
type Row = [string, string, number];
```

```
function processRow(row: Row)
```

Documentación oficial de la interfaz.

<https://www.typescriptlang.org/docs/handbook/interfaces.html>

Examples

Añadir funciones o propiedades a una interfaz existente

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

```
interface JQuery {  
  pluginFunctionThatDoesNothing(): void;  
  
  // create chainable function  
  manipulateDOM(HTMLElement): JQuery;  
}
```

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

Interfaz de clase

Declare variables y métodos `public` escriba en la interfaz para definir cómo otros códigos mecanografiados pueden 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 implements la interfaz.

Interfaz de extensión

Supongamos que tenemos una interfaz:

```

interface IPerson {
    name: string;
    age: number;

    breath(): void;
}

```

Y queremos crear interfaz más específico que tiene las mismas propiedades de la persona, podemos hacerlo mediante el `extends` palabra clave:

```

interface IManager extends IPerson {
    managerId: number;

    managePeople(people: IPerson[]): void;
}

```

Además es posible extender múltiples interfaces.

Uso de interfaces para hacer cumplir tipos

Uno de los beneficios principales de Typescript es que impone tipos de datos de valores que está transmitiendo alrededor de su código para ayudar a prevenir errores.

Digamos que estás haciendo una aplicación de citas para mascotas.

Tiene esta función simple 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 que no escribieron esta función, ignorar que se supone que pasan objetos con 'especies' y 'edad' propiedades Pueden

equivocadamente probar `checkCompatible(petOne.species, petTwo.species)` y luego `checkCompatible(petOne.species, petTwo.species)` para descubrir los errores que se producen cuando la función intenta acceder a `petOne.species.species` o `petOne.species.age`.

Una forma en que podemos evitar que esto suceda es especificando las propiedades que queremos en los parámetros de mascota:

```
checkCompatible(petOne: {species: string, age: number}, petTwo: {species: string, age: number}) {  
    //...  
}
```

En este caso, Typescript se asegurará de que todo lo que se pasa a la función tenga propiedades de 'especie' y 'edad' (está bien si tienen propiedades adicionales), pero esta es una solución poco manejable, incluso con solo dos propiedades especificadas. Con las interfaces, hay una mejor manera!

Primero definimos nuestra interfaz:

```
interface Pet {  
    species: string;  
    age: number;  
    //We can add more properties if we choose.  
}
```

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

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**.

Implementando interfaces genéricas

Crearemos una clase simple para implementar la interfaz de **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 **estatal** .

En nuestro ejemplo, la clase `State` manejará un estado genérico usando `IStatus<T>` . De esta manera, la interfaz `IEvent<T>` también manejará un `IStatus<T>` .

```
const s = new State<IStatus<number>>>();  
  
// The 'code' property is expected to be a number, so:  
s.emit({ code: 200 }); // works  
s.emit({ code: '500' }); // type error  
  
s.getAll().forEach(event => console.log(event.code));
```

Aquí nuestra clase de `State` se escribe 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 de `State` se escribe como `IStatus<Code>` . De esta manera, podemos pasar un tipo más complejo a nuestro método de emisión.

Como puede ver, las interfaces genéricas pueden ser una herramienta muy útil para el código escrito de forma estática.

Usando interfaces para el polimorfismo

La razón principal para usar interfaces para lograr el polimorfismo y proporcionar a los desarrolladores la implementación a su manera en el futuro mediante la implementación de los métodos de la interfaz.

Supongamos que tenemos una interfaz y tres clases:

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

Esta es la interfaz del conector. Ahora lo implementaremos 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. Este es ahora el tipo de `Connector`.

Ahora estamos creando nuestro `System` que tiene un `Connector` componente. 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 el `Connector` es una interfaz, este `System` clase tiene mucha más flexibilidad. Podemos pasar cualquier tipo que haya implementado la interfaz del `Connector`. En el futuro el desarrollador consigue más flexibilidad. Por ejemplo, ahora el desarrollador desea agregar el 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. Hay manera diferente de conectarse. Sin embargo, por lo tanto, ambos han implementado `Type Connector`, ahora son `Type Connector`. Para que podamos pasar cualquiera de ellos a la clase `System` como el parámetro constructor. Esto se llama polimorfismo. El `System` clase ahora no sabe si es Bluetooth / Wifi, incluso podemos agregar otro módulo de comunicación como Inferade, Bluetooth5 y cualquier otro solo implementando la interfaz del `Connector`.

Esto se llama **escribir pato**. `Connector` tipo de `Connector` ahora es dinámico, ya que `doConnect()` es solo un marcador de posición y el desarrollador implementa esto como propio.

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

Implementación implícita y forma del objeto

TypeScript admite interfaces, pero el compilador genera JavaScript, que no lo hace. Por lo tanto, las interfaces se pierden efectivamente en el paso de compilación. Esta es la razón por la que la verificación de tipos en las interfaces se basa en la *forma* del objeto, es decir, si el objeto es compatible con los campos y funciones de la interfaz, y no en si la interfaz se implementa 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);

```

Entonces, incluso si `Ball` no implementa explícitamente `IKickable`, se puede asignar una instancia de `Ball` a (y manipularse como) un `IKickable`, incluso cuando se especifica el tipo.

Lea Interfaces en línea: <https://riptutorial.com/es/typescript/topic/2023/interfaces>

Capítulo 16: Las clases

Introducción

TypeScript, como ECMA Script 6, admite la programación orientada a objetos mediante clases. Esto contrasta con las versiones anteriores de JavaScript, que solo son compatibles con la cadena de herencia basada en prototipos.

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

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

Examples

Clase simple

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

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

En este ejemplo, declaramos una clase simple de `Car`. La clase tiene tres miembros: una `speed` propiedad privada, una `position` propiedad pública y un `move` método público. Tenga en cuenta que cada miembro es público por defecto. Es por eso que `move()` es público, incluso si no usamos 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
```

Herencia básica

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

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

class SelfDrivingCar extends Car {
```

```

    move() {
        // start moving around :-)
        super.move();
        super.move();
    }
}

```

Este ejemplo muestra cómo crear una subclase muy sencilla del `Car` clase utilizando el `extends` palabra clave. La clase `SelfDrivingCar` anula el método `move()` y usa la implementación de la clase base usando `super`.

Constructores

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

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

```

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 se puede resumir 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 mucho 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 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); // access the public property position
```

Accesorios

En este ejemplo, modificamos el ejemplo de "clase simple" para permitir el acceso a la propiedad de `speed` . Los descriptores de acceso de `typescript` nos permiten agregar 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
```

Clases abstractas

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

  // An abstract class can define methods of it's own, or...
  summary(): string {
```



```

        return `${this.manufacturer} makes this machine.`;
    }

    // Require inheriting classes to implement methods
    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 is now 80
console.log(myCar.summary()); // prints "Konda makes this machine."
console.log(myCar.moreInfo()); // prints "This is a car located at 80 and going 70mph!"

```

Las clases abstractas son clases base a partir de las cuales se pueden extender otras clases. No se pueden crear instancias de ellos mismos (es decir, no se **puede** hacer una `new Machine("Konda")`).

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

1. Pueden implementar métodos propios.
2. Pueden definir métodos que las clases heredadas **deben** implementar.

Por esta razón, las clases abstractas pueden considerarse conceptualmente como una **combinación de una interfaz y una clase**.

Monkey parchea una función en una clase existente

A veces es útil poder extender una clase con nuevas funciones. Por ejemplo, supongamos que una cadena se debe convertir en una cadena de caja de camello. Así que debemos decirle a TypeScript que `String` contiene una función llamada `toCamelCase`, que devuelve una `string`.

```

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 esta extensión de `String` está cargada, se puede usar así:

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

Transpilacion

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

Fuente de 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 de JavaScript

Cuando se transpila utilizando 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 está dentro de un **IIFE** .
- Las variables miembro se definen dentro de la `function` clase principal.
- Las propiedades estáticas se agregan directamente al objeto de clase, mientras que las propiedades de instancia se agregan al prototipo.

Lea Las clases en línea: <https://riptutorial.com/es/typescript/topic/1560/las-clases>

Capítulo 17: Mixins

Sintaxis

- La clase BeetleGuy implementa Climbs, Bulletproof {}
- applyMixins (BeetleGuy, [Climbs, Bulletproof]);

Parámetros

Parámetro	Descripción
derivado	La clase que quieres usar como la clase de composición.
baseCtors	Un conjunto de clases que se agregarán a la clase de composición.

Observaciones

Hay tres reglas a tener en cuenta con los mixins:

- Utiliza la palabra clave `implements`, no la palabra clave de `extends` cuando escribes su clase de composición
- Debe tener una firma coincidente para mantener el compilador en silencio (pero no necesita ninguna implementación real, se obtendrá de la mezcla).
- `applyMixins` llamar a `applyMixins` con los argumentos correctos.

Examples

Ejemplo de Mixins

Para crear combinaciones, simplemente declare clases ligeras que se pueden usar como "comportamientos".

```
class Flies {
  fly() {
    alert('Is it a bird? Is it a plane?');
  }
}

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

```
  }
}

class Bulletproof {
  deflect() {
    alert('My wings are a shield of steel.');
```

```
}  
}
```

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 hacer 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];  
            }  
        });  
    });  
}
```

Lea Mixins en línea: <https://riptutorial.com/es/typescript/topic/4727/mixins>

Capítulo 18: Módulos - exportación e importación

Examples

Hola módulo mundial

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

Cargar utilizando índice de directorio

Si el directorio contiene el archivo denominado `index.ts` se puede cargar utilizando solo el nombre del directorio (para `index.ts` nombre de 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"; // load specified elements
import defaultHello from "./hello";    // load default export into name defaultHello
import * as Bundle from "./hello";      // load all exports as Bundle
import {welcome} from "./welcome";      // note index.ts is omitted

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!
```

Exportación / Importación de declaraciones

Cualquier declaración (variable, const, función, clase, etc.) se puede exportar desde un módulo a importar en otro módulo.

Typescript ofrece dos tipos de exportación: nombrada y predeterminada.

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;
```

el cual puede ser importado usando

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

Importación agrupada

Typescript ofrece un método para importar un módulo entero 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
```

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;
    }
}
```

Puedes 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 se pueden reexportar usando una sintaxis más corta

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

Las exportaciones predeterminadas también se pueden exportar, pero no hay una sintaxis corta disponible. Recuerde, solo es posible una exportación predeterminada por módulo.

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

export default Operator;
```

Posible es la reexportación de **importación agrupada**

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

Al reexportar paquetes, las declaraciones pueden ser anuladas 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
```

Lea Módulos - exportación e importación en línea:

<https://riptutorial.com/es/typescript/topic/9054/modulos---exportacion-e-importacion>

Capítulo 19: Publicar archivos de definición de TypeScript

Examples

Incluir archivo de definición con biblioteca en npm

Añadir mecanografía a su package.json

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

Ahora, cuando se haya importado esa biblioteca, el script de tipos cargará el archivo de mecanografía.

Lea [Publicar archivos de definición de TypeScript en línea](https://riptutorial.com/es/typescript/topic/2931/publicar-archivos-de-definicion-de-typescript):

<https://riptutorial.com/es/typescript/topic/2931/publicar-archivos-de-definicion-de-typescript>

Capítulo 20: Tipo de guardias definidos por el usuario

Sintaxis

- `typeof x === "nombre de tipo"`
- `x instanceof TypeName`
- función (foo: any): foo es TypeName {/ * código que devuelve booleano */}

Observaciones

El uso de anotaciones de tipo en TypeScript restringe los tipos posibles con los que tendrá que lidiar su código, pero aún así es común tomar diferentes rutas de código en función del tipo de tiempo de ejecución de una variable.

Las guardas de tipo le permiten escribir código que discrimina en función del tipo de tiempo de ejecución de una variable, mientras se mantiene fuertemente tipado y evita las conversiones (también conocidas como aserciones de tipo).

Examples

Usando 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 is type Dog in this block
    foo.bark();
  }

  if (foo instanceof Cat) {
    // foo is type Cat in this block
    foo.purr();
  }
}
```

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

huellas dactilares

```
woof  
meow
```

a la consola.

Utilizando typeof

`typeof` se usa cuando necesitas distinguir entre tipos `number`, `string`, `boolean` y `symbol`. Otras constantes de cadena no generarán errores, pero tampoco se utilizarán para restringir tipos.

A diferencia de `instanceof`, `typeof` trabajará con una variable de cualquier tipo. En el siguiente ejemplo, `foo` podría escribirse como `number | string` sin problema.

Este código ([pruébalo](#)):

```
function example(foo: any) {  
  if (typeof foo === "number") {  
    // foo is type number in this block  
    console.log(foo + 100);  
  }  
  
  if (typeof foo === "string") {  
    // foo is type string in this block  
    console.log("not a number: " + foo);  
  }  
}  
  
example(23);  
example("foo");
```

huellas dactilares

```
123  
not a number: foo
```

Funciones de protección de tipo

Puedes declarar funciones que sirven como guardias de tipo usando cualquier lógica que desees.

Toman la forma:

```
function functionName(variableName: any): variableName is DesiredType {  
  // body that returns boolean  
}
```

Si la función devuelve true, TypeScript restringirá 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 is type as a string in this block
        console.log("it's a string: " + foo);
    } else {
        // foo is type any in this block
        console.log("don't know what this is! [" + foo + "]");
    }
}

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

Se utiliza un predicado de tipo de función de guardia (el `foo is Bar` en la posición de tipo de retorno de función) en tiempo de compilación para tipos más estrechos, el cuerpo de función se usa en tiempo de ejecución. El tipo de predicado y la función deben coincidir, o su 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 al verificar su cadena de versión.

```
function isjQuery(foo): foo is JQuery {
    // test for jQuery's version string
    return foo.jquery !== undefined;
}

function example(foo) {
    if (isjQuery(foo)) {
        // foo is typed JQuery here
        foo.eq(0);
    }
}
```

Lea [Tipo de guardias definidos por el usuario en línea](#):

<https://riptutorial.com/es/typescript/topic/8034/tipo-de-guardias-definidos-por-el-usuario>

Capítulo 21: Tipografía de ejemplos básicos.

Observaciones

Este es un ejemplo básico que extiende una clase de automóvil genérico y define un método de descripción de automóvil.

Encuentra más ejemplos de TypeScript aquí - [Ejemplos de TypeScript GitRepo](#)

Examples

1 ejemplo básico de herencia de clase usando extended y super keyword

Una clase de automóvil genérico tiene alguna propiedad de automóvil 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 de autos genéricos existentes y agrega 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();
```

2 ejemplo de variable de clase estática: cuente la cantidad de tiempo que se

invoca el método

aquí countInstance es una variable de clase estática

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

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

Lea Tipografía de ejemplos básicos. en línea:

<https://riptutorial.com/es/typescript/topic/7721/tipografia-de-ejemplos-basicos->

Capítulo 22: Tipos básicos de TypeScript

Sintaxis

- `deja variableName: VariableType;`
- `función nombre de función (nombre_parámetro: TipoDeVariable, parámetroDeArteDefault: TipoDeVariable = Parámetro Predeterminado, opcional ¿Parámetro?: Tipo De Variable, ... variarParámetro: VariableType []): Return Type { /*...*/};`

Examples

Booleano

Un booleano representa el tipo de datos más básico en TypeScript, con el propósito de asignar valores de 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;
```

Número

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

```
let pi: number = 3.14;           // base 10 decimal by default
let hexadecimal: number = 0xFF;  // 255 in decimal
```

ECMAScript 2015 permite binario y octal.

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

Cuerda

Tipo de datos textuales:

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

Formación

Una matriz de valores:

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

Enumerar

Un tipo para nombrar un conjunto de valores numéricos:

Los valores numéricos por defecto son 0:

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

Establecer un número inicial predeterminado:

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

o asignar valores:

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

Alguna

Cuando no esté seguro de un tipo, `any` está disponible:

```
let anything: any = 'I am a string';
anything = 5; // but now I am the number 5
```

Vacío

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

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

Tipos de `void` Solo se pueden asignar `null` o `undefined`.

Tupla

Tipo de matriz con tipos conocidos y posiblemente diferentes:

```
let day: [number, string];
day = [0, 'Monday']; // valid
day = ['zero', 'Monday']; // invalid: 'zero' is not numeric
console.log(day[0]); // 0
console.log(day[1]); // Monday
```

```
day[2] = 'Saturday'; // valid: [0, 'Saturday']
day[3] = false;      // invalid: must be union type of 'number | string'
```

Tipos en argumentos de función y valor de retorno. Número

Cuando crea una función en TypeScript, puede especificar el tipo de datos de los argumentos de la función y el tipo de datos para el 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 solo pueden ser números y `(...): number {` significa que el valor de retorno solo puede ser un número

Uso:

```
sum(84 + 76) // will be return 160
```

Nota:

No puedes 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' y error TS2322: Type 'number' is not assignable to type 'string' respectivamente

Tipos en argumentos de función y valor de retorno. Cuerda

Ejemplo:

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

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

Uso:

```
hello('StackOverflow Documentation') // will be return Hello StackOverflow Documentation!
```

Tipos de cuerdas literales

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

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

Cualquier otra cadena dará un error.

```
// Error: Type '"rock"' is not assignable to type '"dog"'.  
// myFavoritePet = "rock";
```

Junto con los alias de tipo y los tipos de unión, obtienes un comportamiento similar a enumeración.

```
type Species = "cat" | "dog" | "bird";  
  
function buyPet(pet: Species, name: string) : Pet { /*...*/ }  
  
buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");  
  
// Error: Argument of type '"rock"' is not assignable to parameter of type '"cat" | "dog" | "bird"'. Type '"rock"' is not assignable to type '"bird"'.  
// buyPet("rock", "Rocky");
```

Los tipos literales de cadena se pueden utilizar para distinguir las 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 /* "dog" as defined above */, "Rocky");  
// dog is from type Dog (dog: Dog)
```

Funcionan bien para los guardias de tipo definido 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 is now from type Cat (pet: Cat)
        pet.eat();
        pet.sleep();
    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)
        pet.eat();
        pet.sing();
        pet.sleep();
    }
}

```

Código de ejemplo completo

```

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

// Error: Type '"rock"' is not assignable to type '"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: Interface 'Rock' incorrectly extends interface 'Pet'. Types of property 'species'
are incompatible. Type '"rock"' is not assignable to type '"cat" | "dog" | "bird"'. Type
'"rock"' is not assignable to type '"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 don't 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}, let's play.`);

    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)

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

        // Error: Type '"bird"' is not assignable to type '"cat"'.
        // pet.type = "bird";

        // Error: Property 'sing' does not exist on type 'Cat'.
        // pet.sing();

    } else if(petIsDog(pet)) {
        // pet is now from type Dog (pet: Dog)

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

    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)

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

let dog = buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");
// dog is from type Dog (dog: Dog)

// Error: Argument of type '"rock"' is not assignable to parameter of type '"cat' | "dog" | "bird"'. Type '"rock"' is not assignable to type '"bird"'.
// buyPet("rock", "Rocky");

playWithPet(dog);
// Output: Hey Rocky, let's play.
//         Rocky eats.
//         Rocky walks.
//         Rocky sleeps.

```

Tipos de interseccion

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();
}

```

const Enum

Un Enum constante 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 el const Enum.

```

// Typescript: A const Enum can be defined like a normal Enum (with start value, specifig values, etc.)
const enum NinjaActivity {
    Espionage,
    Sabotage,
    Assassination
}

// Javascript: But nothing is generated

// Typescript: Except if you use it
let myFavoriteNinjaActivity = NinjaActivity.Espionage;
console.log(myFavoritePirateActivity); // 0

// Javascript: Then only the number of the value is compiled into the code
// var myFavoriteNinjaActivity = 0 /* Espionage */;
// console.log(myFavoritePirateActivity); // 0

// Typescript: The same for the other constant example
console.log(NinjaActivity["Sabotage"]); // 1

// Javascript: Just the number and in a comment the name of the value
// console.log(1 /* "Sabotage" */); // 1

// Typescript: But without the object none runtime access is possible
// Error: A const enum member can only be accessed using a string literal.
// console.log(NinjaActivity[myFavoriteNinjaActivity]);

```

Para comparación, un Enum normal

```

// Typescript: A normal Enum

```

```

enum PirateActivity {
    Boarding,
    Drinking,
    Fencing
}

// Javascript: The Enum after the compiling
// var PirateActivity;
// (function (PirateActivity) {
//     PirateActivity[PirateActivity["Boarding"] = 0] = "Boarding";
//     PirateActivity[PirateActivity["Drinking"] = 1] = "Drinking";
//     PirateActivity[PirateActivity["Fencing"] = 2] = "Fencing";
// })(PirateActivity || (PirateActivity = {}));

// Typescript: A normale use of this Enum
let myFavoritePirateActivity = PirateActivity.Boarding;
console.log(myFavoritePirateActivity); // 0

// Javascript: Looks quite similar in Javascript
// var myFavoritePirateActivity = PirateActivity.Boarding;
// console.log(myFavoritePirateActivity); // 0

// Typescript: And some other normale use
console.log(PirateActivity["Drinking"]); // 1

// Javascript: Looks quite similar in Javascript
// console.log(PirateActivity["Drinking"]); // 1

// Typescript: At runtime, you can access an normal enum
console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"

// Javascript: And it will be resolved at runtime
// console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"

```

Lea Tipos básicos de TypeScript en línea: <https://riptutorial.com/es/typescript/topic/2776/tipos-basicos-de-typescript>

Capítulo 23: tsconfig.json

Sintaxis

- Utiliza el formato de archivo JSON
- También puede aceptar comentarios de estilo JavaScript

Observaciones

Visión general

La presencia de un archivo `tsconfig.json` en un directorio indica que el directorio es la raíz de un proyecto de TypeScript. El archivo `tsconfig.json` especifica los archivos raíz y las opciones del compilador necesarias para compilar el proyecto.

Utilizando tsconfig.json

- Al invocar `tsc` sin archivos de entrada, en cuyo caso el compilador busca el archivo `tsconfig.json` que comienza en el directorio actual y continúa en la cadena del directorio principal.
- Al invocar `tsc` sin archivos de entrada y una opción de línea de comando `--project` (o simplemente `-p`) que especifica la ruta de un directorio que contiene un archivo `tsconfig.json`. Cuando los archivos de entrada se especifican en la línea de comando, los archivos `tsconfig.json` son

Detalles

La propiedad `"compilerOptions"` se puede omitir, en cuyo caso se utilizan los valores predeterminados del compilador. Vea nuestra lista completa de [opciones de compilador](#) compatibles.

Si no hay una propiedad de `"files"` en un `tsconfig.json`, el compilador incluye de manera predeterminada todos los archivos de TypeScript (`*.ts` o `*.tsx`) en el directorio y subdirectorios que lo contienen. Cuando una propiedad de `"archivos"` está presente, solo se incluyen los archivos especificados.

Si se especifica la propiedad `"exclude"`, el compilador incluye todos los archivos de TypeScript (`*.ts` o `*.tsx`) en el directorio y subdirectorios que contienen, excepto aquellos archivos o carpetas que están excluidos.

La propiedad `"files"` no se puede utilizar junto con la propiedad `"excluir"`. Si se especifican ambos, entonces la propiedad `"archivos"` tiene prioridad.

También se incluyen los archivos a los que hacen referencia los especificados en la propiedad "files" . De manera similar, si a otro archivo A se hace referencia a un archivo B.ts, entonces no se puede excluir a B.ts a menos que también se especifique el archivo de referencia A.ts en la lista "excluir".

Se `tsconfig.json` archivo `tsconfig.json` esté completamente vacío, lo que compila todos los archivos en el directorio que contiene y los subdirectorios con las opciones de compilador predeterminadas.

Las opciones de compilador especificadas en la línea de comando anulan las especificadas en el archivo `tsconfig.json`.

Esquema

El esquema se puede encontrar en: <http://json.schemastore.org/tsconfig>

Examples

Crear proyecto de 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.

La inicialización de un proyecto de TypeScript, o mejor poner el archivo `tsconfig.json`, se puede hacer a través del siguiente comando:

```
tsc --init
```

A partir de TypeScript v2.3.0 y superior, se creará el siguiente `tsconfig.json` de forma predeterminada:

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",                                /* Specify ECMAScript target version: 'ES3'
    (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'. */
    "module": "commonjs",                          /* Specify module code generation: 'commonjs',
    'amd', 'system', 'umd' or 'es2015'. */
    // "lib": [],                                   /* Specify library files to be included in the
    compilation: */
    // "allowJs": true,                             /* Allow javascript files to be compiled. */
    // "checkJs": true,                             /* Report errors in .js files. */
    // "jsx": "preserve",                           /* Specify JSX code generation: 'preserve',
    'react-native', or 'react'. */
    // "declaration": true,                         /* Generates corresponding '.d.ts' file. */
    // "sourceMap": true,                           /* Generates corresponding '.map' file. */
    // "outFile": "./",                             /* Concatenate and emit output to single file.
    */
    // "outDir": "./",                             /* Redirect output structure to the directory.
    */
  }
}
```

```

    // "rootDir": "./", // Specify the root directory of input files.
Use to control the output directory structure with --outDir. */
    // "removeComments": true, // Do not emit comments to output. */
    // "noEmit": true, // Do not emit outputs. */
    // "importHelpers": true, // Import emit helpers from 'tslib'. */
    // "downlevelIteration": true, // Provide full support for iterables in 'for-
of', spread, and destructuring when targeting 'ES5' or 'ES3'. */
    // "isolatedModules": true, // Transpile each file as a separate module
(similar to 'ts.transpileModule'). */

    /* Strict Type-Checking Options */
    "strict": true // Enable all strict type-checking options. */
    // "noImplicitAny": true, // Raise error on expressions and declarations
with an implied 'any' type. */
    // "strictNullChecks": true, // Enable strict null checks. */
    // "noImplicitThis": true, // Raise error on 'this' expressions with an
implied 'any' type. */
    // "alwaysStrict": true, // Parse in strict mode and emit "use strict"
for each source file. */

    /* Additional Checks */
    // "noUnusedLocals": true, // Report errors on unused locals. */
    // "noUnusedParameters": true, // Report errors on unused parameters. */
    // "noImplicitReturns": true, // Report error when not all code paths in
function return a value. */
    // "noFallthroughCasesInSwitch": true, // Report errors for fallthrough cases in switch
statement. */

    /* Module Resolution Options */
    // "moduleResolution": "node", // Specify module resolution strategy: 'node'
(Node.js) or 'classic' (TypeScript pre-1.6). */
    // "baseUrl": "./", // Base directory to resolve non-absolute module
names. */
    // "paths": {}, // A series of entries which re-map imports to
lookup locations relative to the 'baseUrl'. */
    // "rootDirs": [], // List of root folders whose combined content
represents the structure of the project at runtime. */
    // "typeRoots": [], // List of folders to include type definitions
from. */
    // "types": [], // Type declaration files to be included in
compilation. */
    // "allowSyntheticDefaultImports": true, // Allow default imports from modules with no
default export. This does not affect code emit, just typechecking. */

    /* Source Map Options */
    // "sourceRoot": "./", // Specify the location where debugger should
locate TypeScript files instead of source locations. */
    // "mapRoot": "./", // Specify the location where debugger should
locate map files instead of generated locations. */
    // "inlineSourceMap": true, // Emit a single file with source maps instead
of having a separate file. */
    // "inlineSources": true, // Emit the source alongside the sourcemaps
within a single file; requires '--inlineSourceMap' or '--sourceMap' to be set. */

    /* Experimental Options */
    // "experimentalDecorators": true, // Enables experimental support for ES7
decorators. */
    // "emitDecoratorMetadata": true, // Enables experimental support for emitting
type metadata for decorators. */
  }
}

```

La mayoría, si no todas, las opciones se generan automáticamente con solo las necesidades básicas que quedan sin comentarios.

Las versiones anteriores de TypeScript, como por ejemplo v2.0.x y versiones inferiores, generarían un `tsconfig.json` como este:

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

compileOnSave

Establecer una propiedad de nivel superior `compileOnSave` señala al IDE para generar todos los archivos para un `tsconfig.json` dado al guardar.

```
{
  "compileOnSave": true,
  "compilerOptions": {
    ...
  },
  "exclude": [
    ...
  ]
}
```

Esta función está disponible desde TypeScript 1.8.4 y posteriores, pero debe ser compatible directamente con IDE. Actualmente, los ejemplos de IDE soportados son:

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

Comentarios

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

```
//Leading comment
{
  "compilerOptions": {
    //this is a line comment
    "module": "commonjs", //eol line comment
    "target" /*inline block*/ : "es5",
    /* This is a
    block
    comment */
  }
}
```

```
/* trailing comment */
```

Configuración para menos errores de programación.

Hay configuraciones muy buenas para forzar la escritura y obtener errores más útiles que no están activados de forma predeterminada.

```
{
  "compilerOptions": {

    "alwaysStrict": true, // Parse in strict mode and emit "use strict" for each source file.

    // If you have wrong casing in referenced files e.g. the filename is Global.ts and you
    // have a /// <reference path="global.ts" /> to reference this file, then this can cause to
    // unexpected errors. Visite: http://stackoverflow.com/questions/36628612/typescript-transpiler-casing-issue
    "forceConsistentCasingInFileNames": true, // Disallow inconsistently-cased references to
    // the same file.

    // "allowUnreachableCode": false, // Do not report errors on unreachable code. (Default:
    // False)
    // "allowUnusedLabels": false, // Do not report errors on unused labels. (Default: False)

    "noFallthroughCasesInSwitch": true, // Report errors for fall through cases in switch
    // statement.
    "noImplicitReturns": true, // Report error when not all code paths in function return a
    // value.

    "noUnusedParameters": true, // Report errors on unused parameters.
    "noUnusedLocals": true, // Report errors on unused locals.

    "noImplicitAny": true, // Raise error on expressions and declarations with an implied
    // "any" type.
    "noImplicitThis": true, // Raise error on this expressions with an implied "any" type.

    "strictNullChecks": true, // The null and undefined values are not in the domain of every
    // type and are only assignable to themselves and any.

    // To enforce this rules, add this configuration.
    "noEmitOnError": true // Do not emit outputs if any errors were reported.
  }
}
```

¿No es suficiente? Si usted es un programador de hardware y desea más, puede que le interese comprobar los archivos de TypeScript con tslint antes de compilarlos con tsc. Verifique cómo [configurar tslint para un código aún más estricto](#) .

preserveConstEnums

Typescript soporta enumerables de constant, declarados a través de `const enum` .

Por lo general, esto solo es sintaxis de azúcar, ya que los enumerados de constant están incluidos en JavaScript compilado.

Por ejemplo el siguiente código

```
const enum Tristate {
    True,
    False,
    Unknown
}

var something = Tristate.True;
```

compila a

```
var something = 0;
```

Aunque el rendimiento se beneficia de la incorporación, es posible que prefiera mantener los enumerados incluso si es costoso (es decir, puede desear legibilidad en el código de desarrollo). Para hacerlo, debe configurar **tsconfig.json** la `preserveConstEnums` en el `compilerOptions` para que sea `true`.

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

De esta manera, el ejemplo anterior se compilaría como cualquier otro enum, como se muestra en el siguiente fragmento de código.

```
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
```

Lea **tsconfig.json** en línea: <https://riptutorial.com/es/typescript/topic/4720/tsconfig-json>

Capítulo 24: TSLint - asegurando la calidad y consistencia del código

Introducción

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

Examples

Configuración básica de tslint.json

Esta es una configuración básica de `tslint.json` que

- evita el uso de `any`
- requiere llaves para las declaraciones `if / else / for / do / while`
- requiere comillas dobles (`"`) para ser utilizadas para cadenas

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

Configuración para menos errores de programación.

Este ejemplo de `tslint.json` contiene un conjunto de configuración para imponer más tipificaciones, detectar errores comunes o construcciones confusas que son propensas a producir errores y seguir más las [Pautas de codificación para los contribuyentes de TypeScript](#).

Para hacer cumplir estas reglas, incluya `tslint` en su proceso de compilación y verifique su código antes de compilarlo con `tsc`.

```
{
  "rules": {
    // TypeScript Specific
    "member-access": true, // Requires explicit visibility declarations for class members.
    "no-any": true, // Disallows usages of any as a type declaration.
    // Functionality
    "label-position": true, // Only allows labels in sensible locations.
    "no-bitwise": true, // Disallows bitwise operators.
    "no-eval": true, // Disallows eval function invocations.
    "no-null-keyword": true, // Disallows use of the null keyword literal.
    "no-unsafe-finally": true, // Disallows control flow statements, such as return,
    continue, break and throws in finally blocks.
    "no-var-keyword": true, // Disallows usage of the var keyword.
  }
}
```

```

    "radix": true, // Requires the radix parameter to be specified when calling parseInt.
    "triple-equals": true, // Requires === and !== in place of == and !=.
    "use-isnan": true, // Enforces use of the isNaN() function to check for NaN references
instead of a comparison to the NaN constant.
    // Style
    "class-name": true, // Enforces PascalCased class and interface names.
    "interface-name": [ true, "never-prefix" ], // Requires interface names to begin with a
capital 'I'
    "no-angle-bracket-type-assertion": true, // Requires the use of as Type for type
assertions instead of <Type>.
    "one-variable-per-declaration": true, // Disallows multiple variable definitions in the
same declaration statement.
    "quotemark": [ true, "double", "avoid-escape" ], // Requires double quotes for string
literals.
    "semicolon": [ true, "always" ], // Enforces consistent semicolon usage at the end of
every statement.
    "variable-name": [true, "ban-keywords", "check-format", "allow-leading-underscore"] //
Checks variable names for various errors. Disallows the use of certain TypeScript keywords
(any, Number, number, String, string, Boolean, boolean, undefined) as variable or parameter.
Allows only camelCased or UPPER_CASED variable names. Allows underscores at the beginning
(only has an effect if "check-format" specified).
  }
}

```

Usando un conjunto de reglas predefinido como predeterminado

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

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

`tslint:latest` extensión `tslint: recomendado` y se actualiza continuamente para incluir la configuración de las últimas reglas en cada versión de TSLint. El uso de esta configuración puede introducir cambios de última hora en versiones menores a medida que se habilitan nuevas reglas que causan fallas de pelusa en su código. Cuando TSLint alcanza un gran golpe de versión, `tslint: recomendado` se actualizará para que sea idéntico a `tslint: más reciente`.

[Documentos y código fuente de un conjunto de reglas predefinido.](#)

Así que uno puede simplemente usar:

```

{
  "extends": "tslint:recommended"
}

```

Para tener una configuración de partida sensible.

Entonces se pueden sobrescribir las reglas de ese ajuste preestablecido a través de las `rules`, por ejemplo, para los desarrolladores de nodos, tenía sentido establecer `no-console` en `false`


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

Instalación y configuración

Para instalar el comando [tslint](#) run

```
npm install -g tslint
```

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

```
tslint --init
```

Para comprobar el archivo de posibles errores en el comando de ejecución de archivo

```
tslint filename.ts
```

Conjuntos de Reglas TSLint

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

El generador de Yeoman es compatible con todos estos ajustes preestablecidos y puede extenderse también:

- [generador-tslint](#)

Lea [TSLint - asegurando la calidad y consistencia del código en línea:](#)

<https://riptutorial.com/es/typescript/topic/7457/tslint---asegurando-la-calidad-y-consistencia-del-codigo>

Capítulo 25: TypeScript con AngularJS

Parámetros

Nombre	Descripción
controllerAs	es un nombre de alias al que se pueden asignar variables o funciones. @see: https://docs.angularjs.org/guide/directive
\$inject	Lista de inyección de dependencias, se resuelve de forma angular y pasa como un argumento a las funciones constructor.

Observaciones

Al realizar la directiva en TypeScript, tenga en cuenta el poder de este lenguaje de tipo personalizado e interfaces que puede crear. Esto es extremadamente útil cuando se desarrollan aplicaciones enormes. La finalización del código soportado por muchos IDE le mostrará el valor posible por el tipo correspondiente con el que está trabajando, por lo que hay mucho más que debe tener en cuenta (en comparación con VanillaJS).

"Código contra interfaces, no implementaciones"

Examples

Directiva

```
interface IMyDirectiveController {
    // specify exposed controller methods and properties here
    getUrl(): string;
}

class MyDirectiveController implements IMyDirectiveController {

    // Inner injections, per each directive
    public static $inject = ['$location', 'toaster'];

    constructor(private $location: ng.ILocationService, private toaster: any) {
        // $location and toaster are now properties of the controller
    }

    public getUrl(): string {
        return this.$location.url(); // utilize $location to retrieve the URL
    }
}

/*
 * Outer injections, for run once controll.
 * For example we have all templates in one value, and we wan't to use it.
 */
```

```

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,
];

// Using slug naming across the projects simplifies change of the directive name
myDirective.prototype.slug = "myDirective";

// You can place this in some bootstrap file, or have them at the same file
angular.module("myApp").
  directive(myDirective.prototype.slug, myDirective);

```

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,
  };
}

// Using slug naming across the projects simplifies change of the directive name
myDirective.prototype.slug = "myDirective";

// You can place this in some bootstrap file, or have them at the same file
angular.module("myApp").

```

```
directive(myDirective.prototype.slug, [
    Templates.prototype.slug,
    myDirective
]);
```

Componente

Para una transición más fácil a Angular 2, se recomienda usar `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);
```

componentes / 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>
```

controlador / 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 {
        // implementation..
    }
}
```

```
}  
}
```

servicios / MyModuleService.ts

```
export class MyModuleService {  
  public static readonly $inject: string[] = [];  
  
  constructor() {  
  }  
  
  public doSomething(): void {  
    // do something  
  }  
}
```

somewhere.html

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

Lea TypeScript con AngularJS en línea: <https://riptutorial.com/es/typescript/topic/6569/typescript-con-angularjs>

Capítulo 26: TypeScript con SystemJS

Examples

Hola mundo en el navegador con SystemJS

Instalar systemjs y plugin-typescript

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

NOTA: esto instalará el compilador mecanografiado 2.0.0 que aún no se ha 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 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 verificación de tipos, elimine el `loader: "plugin-typescript"` y `typescriptOptions` de `config.js`. También tenga en cuenta que nunca verificará 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

```

Constrúyelo para la producción.

```

npm install systemjs-builder

```

Crear el 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` desde `hello.ts`

```

node build.js

```

Utilízalo en producción.

Simplemente cargue hello.js con una etiqueta de script antes del primer uso

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>
```

Lea TypeScript con SystemJS en línea: <https://riptutorial.com/es/typescript/topic/6664/typescript-con-systemjs>

Capítulo 27: Typescript-installation-typescript-and-running-the-typescript-compiler-tsc

Introducción

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

Examples

Pasos.

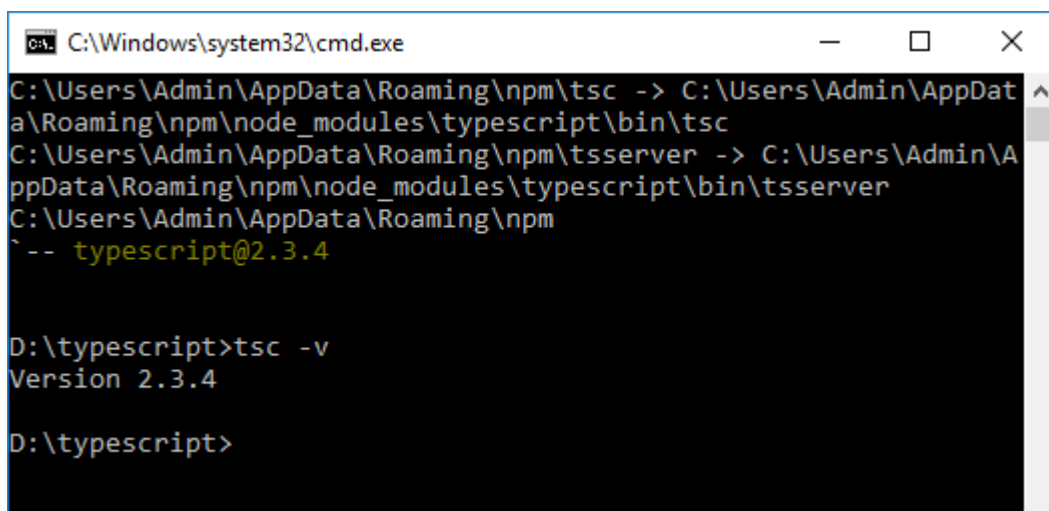
Instalar Typescript y ejecutar el compilador de typScript.

Para instalar Typescript Comiler

```
npm install -g typescript
```

Para consultar con la versión mecanografiada.

```
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

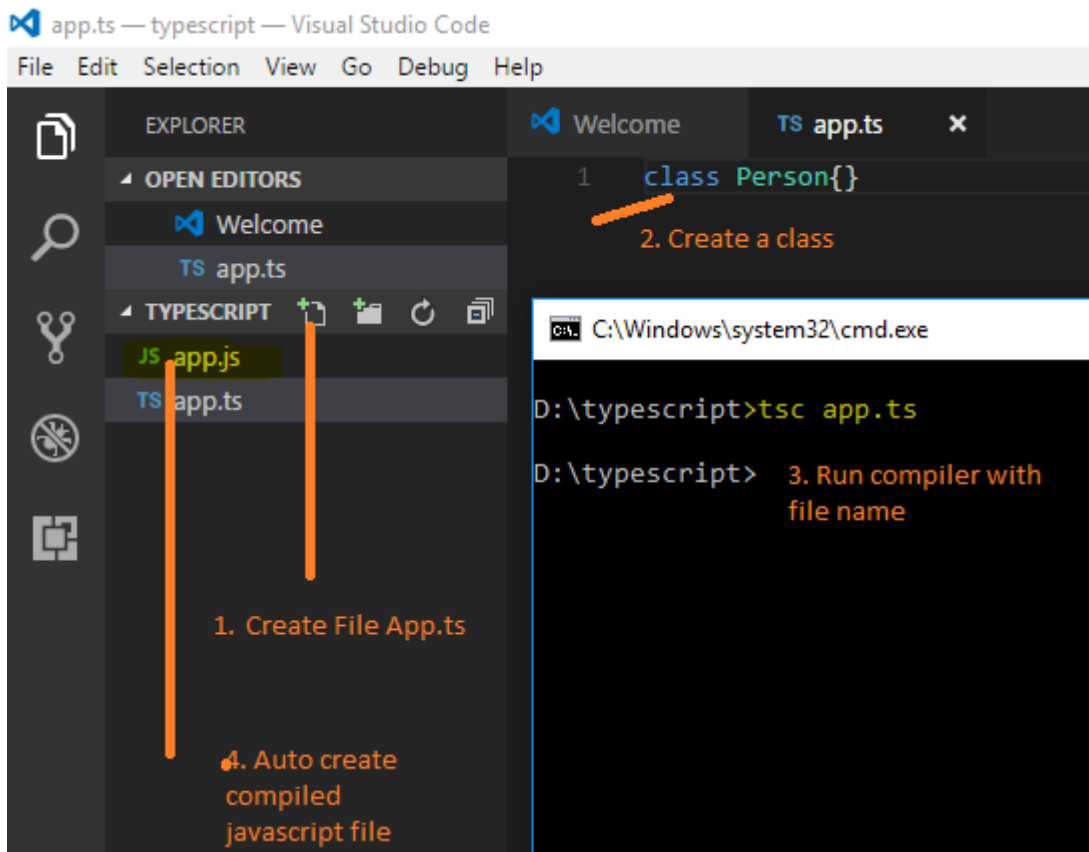
D:\typescript>
```

Descargar Visual Studio Code para Linux / Windows

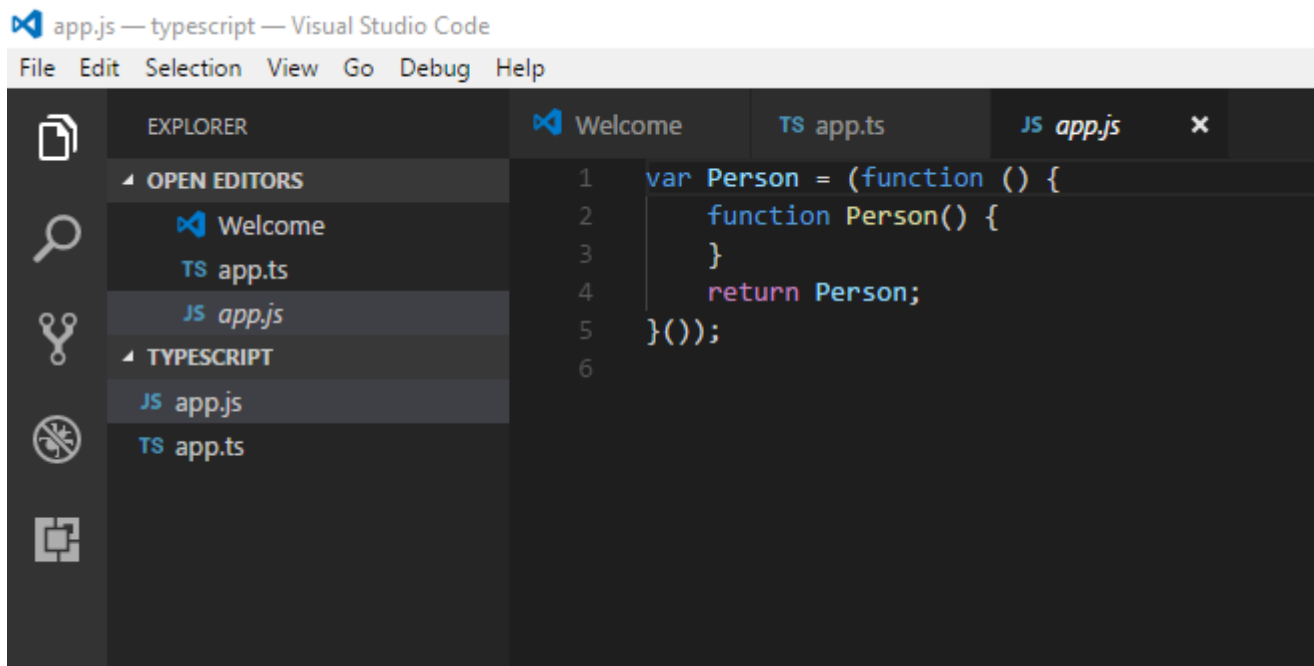
[Enlace de descarga de código visual](#)

1. Abrir código de Visual Studio
2. Abra Same Folde donde haya instalado el compilador Typescript

3. Agregar archivo haciendo clic en el ícono más en el panel izquierdo
4. Crear una clase básica.
5. Compila tu tipo de archivo de script y genera un resultado.



Ver el resultado en javascript compilado de código escrito mecanografiado.



Gracias.

Lea [Typescript-installation-typescript-and-running-the-typescript-compiler-tsc](https://riptutorial.com/es/typescript/topic/10503/typescript-installation-typescript-and-running-the-) en línea:
<https://riptutorial.com/es/typescript/topic/10503/typescript-installation-typescript-and-running-the->

Capítulo 28: Usando Typescript con React (JS & native)

Examples

Componente ReactJS escrito en Typescript

Puede utilizar los componentes de ReactJS fácilmente en TypeScript. Simplemente cambie 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 característica principal de TypeScript (comprobación de tipos estática), debe 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, mira [aquí](#)

2) Agregar apoyos y interfaces de estado:

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

interface State {
  //empty in our case
}

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

```
// TypeScript will allow you to create without the optional parameter
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
// But it does check if you pass in an optional parameter of the wrong type
ReactDOM.render(<HelloMessage name="Sebastian" optionalParam='foo' />, mountNode);
```

Ahora TypeScript mostrará un error si el programador se olvida de aprobar las propuestas. O si intentas pasar props que no están definidos en la interfaz.

Mecanografiado y reaccionar y paquete web

Instalación de mecanografía, typings y webpack globalmente.

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

Instalación de cargadores y vinculación de escritura de tipos

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

La vinculación de TypeScript permite que ts-loader use su instalación global de TypeScript en lugar de necesitar una copia local separada de [documentos de escritura](#).

instalando archivos `.d.ts` con mecanografía 2.x

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

instalando archivos `.d.ts` con mecanografía 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"
  }
}
```

`webpack.config.js` configuración `webpack.config.js`

```
module.exports = {
  entry: "<path to entry point>", // for example ./src/helloMessage.tsx
  output: {
    filename: "<path to bundle file>", // for example ./dist/bundle.js
  },

  // Enable sourcemaps for debugging webpack's output.
```

```

devtool: "source-map",

resolve: {
  // Add '.ts' and '.tsx' as resolvable extensions.
  extensions: [".", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
},

module: {
  loaders: [
    // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
    {test: /\.tsx?$/, loader: "ts-loader"}
  ],

  preLoaders: [
    // All output '.js' files will have any sourcemaps re-processed by 'source-map-
loader'.
    {test: /\.js$/, loader: "source-map-loader"}
  ]
},

// When importing a module whose path matches one of the following, just
// assume a corresponding global variable exists and use that instead.
// This is important because it allows us to avoid bundling all of our
// dependencies, which allows browsers to cache those libraries between builds.
externals: {
  "react": "React",
  "react-dom": "ReactDOM"
},
};

```

finalmente ejecute `webpack` o `webpack -w` (para el modo de visualización)

Nota : React y ReactDOM están marcados como externos

Lea Usando Typescript con React (JS & native) en línea:

<https://riptutorial.com/es/typescript/topic/1835/usando-typescript-con-react--js--amp--native->

Capítulo 29: Usando Typescript con RequireJS

Introducción

RequireJS es un archivo JavaScript y un cargador de módulos. Está optimizado para su uso en el navegador, pero se puede usar en otros entornos de 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, e incluir un fragmento de código en cualquier archivo HTML. El compilador traducirá las importaciones de la sintaxis de TypeScript al formato RequireJS.

Examples

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>
```

Ejemplo de tsconfig.json para compilar para ver la carpeta usando el estilo de importación requireJS.

```
{
  "module": "amd",    // Using AMD module code generator which works with requireJS
  "rootDir": "./src", // Change this to your source folder
  "outDir": "./view",
  ...
}
```

Lea Usando Typescript con RequireJS en línea:

<https://riptutorial.com/es/typescript/topic/10773/usando-typescript-con-requirejs>

Capítulo 30: Usando TypeScript con webpack

Examples

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" // if you want to use react jsx
  }
}
```

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

  // Enable sourcemaps for debugging webpack's output.
  devtool: "source-map",

  resolve: {
    // Add '.ts' and '.tsx' as resolvable extensions.
    extensions: [".", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
  },

  module: {
    loaders: [
      // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
      {test: /\.tsx?$/, loader: "ts-loader"}
    ],

    preLoaders: [
      // All output '.js' files will have any sourcemaps re-processed by 'source-map-
      loader'.
      {test: /\.js$/, loader: "source-map-loader"}
    ]
  },
  /*****
   * If you want to use react *
   *****/

  // When importing a module whose path matches one of the following, just
  // assume a corresponding global variable exists and use that instead.
  // This is important because it allows us to avoid bundling all of our
  // dependencies, which allows browsers to cache those libraries between builds.
```



```
// externals: {  
//   "react": "React",  
//   "react-dom": "ReactDOM"  
// },  
};
```

Lea Usando TypeScript con webpack en línea:

<https://riptutorial.com/es/typescript/topic/2024/usando-typescript-con-webpack>

Creditos

S. No	Capítulos	Contributors
1	Empezando con TypeScript	2426021684 , Alec Hansen , Blackus , BrunoLM , cdbajorin , ChanceM , Community , danvk , Florian Hämmerle , Fylax , goenning , islandman93 , jengeb , Joshua Breeden , k0pernikus , Kiloku , KnottytOmo , Kuba Beránek , Lekhnath , Matt Lishman , Mikhail , mleko , RationalDev , Roy Dictus , Saiful Azad , Sam , samAlvin , Wasabi Fan , zigzag
2	¿Por qué y cuándo usar TypeScript?	danvk
3	Arrays	Udlei Nati
4	Cómo utilizar una biblioteca javascript sin un archivo de definición de tipo	Bruno Krebs , Kevin Montrose
5	Configure el proyecto de mecanografía para compilar todos los archivos en mecanografía	Rahul
6	Controles nulos estrictos	bnieland , danvk , JKillian , Yaroslav Admin
7	Decorador de clase	bruno , Remo H. Jansen , Stefan Rein
8	Depuración	Peopleware
9	Enums	dimitrisli , Florian Hämmerle , Kevin Montrose , smnbbrv
10	Examen de la unidad	James Monger , leonidv , Louie Bertoncin , Matthew Harwood , mleko
11	Funciones	br4d , hansmaad , islandman93 , KnottytOmo , muetzerich , SilentLupin , Slava Shpitalny
12	Genéricos	danvk , hansmaad , KnottytOmo , Mathias Rodriguez , Muhammad Awais , Slava Shpitalny , Taytay

13	Importando bibliotecas externas	2426021684 , Almond , artem , Blackus , Brutus , Dean Ward , duplicator , Harry , islandman93 , JKillian , Joel Day , KnottytOmo , lefb766 , Rajab Shakirov , Slava Shpitalny , tuvokki
14	Integración con herramientas de construcción	Alex Filatov , BrunoLM , Dan , duplicator , John Ruddell , mleko , Protectator , smnbbvr , void
15	Interfaces	ABabin , Aminadav , Aron , artem , Cobus Kruger , Fabian Lauer , islandman93 , Joshua Breeden , Paul Boutes , Robin , Saiful Azad , Slava Shpitalny , Sunnyok
16	Las clases	adamboro , apricity , Cobus Kruger , Equiman , hansmaad , James Monger , Jeff Huijsmans , Justin Niles , KnottytOmo , Robin
17	Mixins	Fenton
18	Módulos - exportación e importación	mleko
19	Publicar archivos de definición de TypeScript	2426021684
20	Tipo de guardias definidos por el usuario	Kevin Montrose
21	Tipografía de ejemplos básicos.	vashishth
22	Tipos básicos de TypeScript	duplicator , Fenton , Fylax , Magu , Mikhail , Moriarty , RationalDev
23	tsconfig.json	bnieland , Fylax , goenning , Magu , Moriarty , user3893988
24	TSLint - asegurando la calidad y consistencia del código	Alex Filatov , James Monger , k0pernikus , Magu , mleko
25	TypeScript con AngularJS	Chic , Roman M. Koss , Stefan Rein
26	TypeScript con SystemJS	artem
27	Typescript-installation-	Rahul

	typescript-and-running-the-typescript-compiler-tsc	
28	Usando Typescript con React (JS & native)	Aleh Kashnikau , irakli khitarishvili , islandman93 , Rajab Shakirov , tBX
29	Usando Typescript con RequireJS	lilezek
30	Usando TypeScript con webpack	BrunoLM , irakli khitarishvili , John Ruddell