



# TypeScript


## Links

<https://www.typescripttutorial.net/>

 [Tutorial de TypeScript completo.](#)

### TypeScript

Junghoo Cho Superset of JavaScript (a.k.a. JavaScript++) New features: types, interfaces, decorators, ... All additional TypeScript features are strictly optional and are not required Any JavaScript code is also a TypeScript code! Transpilation function `hello(greeting: string): string { return "Hello, " + greeting + "!"; }` `hello([0, 1, 2]);`

 <http://oak.cs.ucla.edu/classes/cs144/slides/typescript.html#/7>

 [Slides con el mismo contenido pero en mayor brevedad.](#)

### TypeScript on Exercism

`export class HelloWorld { static hello(name = 'World'): string { return `Hello, ${name}!` } }` Get better at programming through fun, rewarding coding exercises that test your understanding of

 <https://exercism.org/tracks/typescript>



 [Track de Exercism de TypeScript.](#)

## Índice:

[Links](#)

[Índice:](#)

[Qué es TypeScript](#)

[Set-up](#)

[Instalar el compilador de TypeScript](#)

[Uso básico](#)

[Tipos](#)

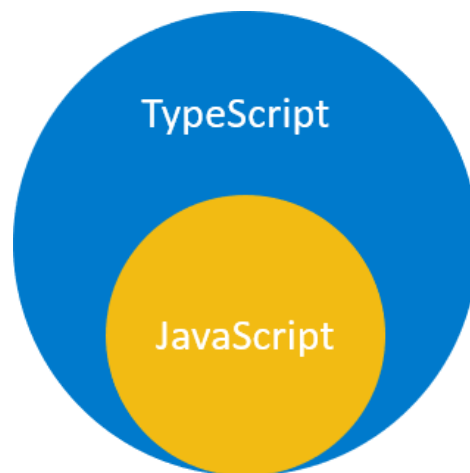
[Inferencia de tipos](#)

- [Aviso sobre los tipos](#)
- [Objetos](#)
- [Tuplas](#)
- [Enum](#)
  - [Cuando usar un enum](#)
- [Tipo any](#)
- [Tipo void](#)
- [Tipo never](#)
- [Unión de tipos](#)
- [Aliases](#)
- [Strings literales](#)
- [Funciones](#)
  - [En caso de funciones flecha](#)
- [Parámetros opcionales](#)
- [Parámetros 'rest'](#)
- [Clases](#)
  - [Modificadores de acceso](#)
  - [Readonly](#)
  - [Getters & Setters](#)
  - [Herencia](#)
  - [Métodos estáticos](#)
  - [Clases abstractas](#)
- [Interfaces](#)
  - [Tipos de funciones](#)
  - [Tipos de clases](#)
  - [Extender interfaces](#)
    - [Interfaces extendiendo clases](#)
- [Intersección de tipos](#)
- [Guardias](#)
- [Casteo de tipos](#)
- [Aserción de tipos](#)
- [Plantillas](#)
  - [Plantillas con varios tipos](#)
  - [Beneficios a usar plantillas](#)
  - [Restricciones de plantillas](#)
  - [Clases plantilla](#)
  - [Interfaces plantilla](#)
- [Módulos en TypeScript](#)
- [Automatizar el desarrollo](#)

# Qué es TypeScript

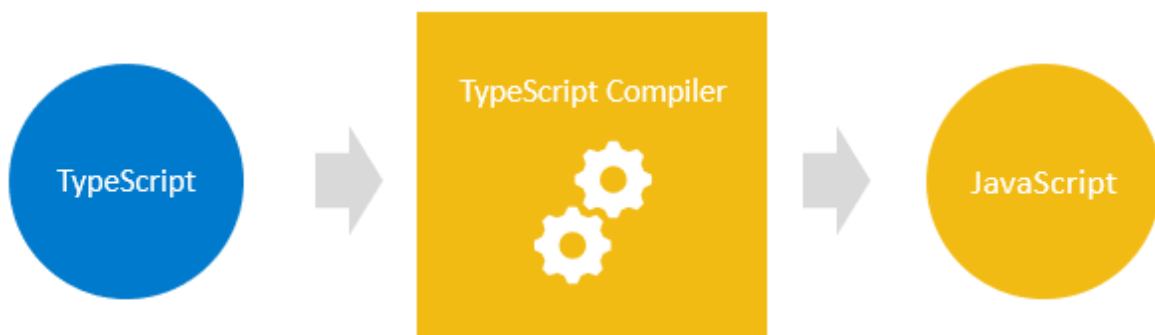
Básicamente, JavaScript pero más completo, con tipado, y te evita un gran número de errores mediante el uso de un compilador. También incluye herramientas adicionales como las `interfaces`, o las `guardias`.

TypeScript es un superset de JavaScript. Todo código de JS también es un código de TS. La extensión de TS es `.ts`. Lo que hace TS es añadir sintaxis a JS, y algunos elementos adicionales que veremos en mayor profundidad.



El flujo para generar el ejecutable a partir de un código en TS es el siguiente:

1. Compila el código de TS (comprueba que está bien escrito)
2. Lo transforma a un programa con el mismo nombre que ejecuta las mismas funcionalidades que tu programa en TS.



**Básicamente, te ayuda a evitar errores.**

Adicionalmente, trae funcionalidades que son **candidatas** de ser nuevas funcionalidades de JS para ser usadas directamente.

## Set-up

Necesitarás:

- Node.js
- TypeScript compiler - un módulo de Node.js que compila TS a JS. Puedes instalar el módulo `ts-node`.
- Un editor de texto
  - Adicionalmente, si usas Visual Studio Code (VSC) puedes instalar la extensión **Live Server** para hacer las cosas más rápidas si programas de cara a buscador.

## Instalar el compilador de TypeScript

En tu terminal de comandos favorita:

```
npm install -g typescript
```

Después de instalar, comprueba la versión:

```
tsc --v
```

Debería devolver una versión con el siguiente formato:

```
Version 4.0.2
```



En Windows es posible que te de un error indicando que no reconoce 'tsc'. Para arreglarlo, añade `C:\Users\<user>\AppData\Roaming\npm` a la variable `PATH`.

Donde <user> es tu nombre de usuario.

Para instalar el llamado `ts-node`, ejecuta esto:

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

## Uso básico

1. Escribe tu programa en TS.

2. Ejecuta `tsc app.ts`, donde `app.ts` es tu fichero en TS.
3. Si todo fue bien un fichero llamado `app.js` (depende del nombre de tu fichero `.ts`) aparecerá.
4. Ejecuta el programa js: `node app.js`.
5. **Nótese:** Si has instalado `ts-node`, puedes hacer la compilación y ejecución en un solo comando: `ts-node app.ts`.

## Tipos

Un punto muy interesante de TS es que nos trae tipos, `types` a la “caja de herramientas”. Su uso es muy simple, y aquí un resumen simple de los tipos:

```
let a: number = 1;
let b: string = "A";
let c: boolean = true;
let d: number[] = [1, 2]; // array
let e: [number, string] = [1, "A"]; // tuple
let f: (number | string) = 1; // or "A" // union
let g: any = "A"; // or 1, [1, 2], ... // any
let h: void = undefined; // or null // void
let i: never; // nothing can be assigned // never

let j: {x: number, y: string} = {x:1, y:"A"}; // object

function hello(name: string): void { // function
    console.log(`Hello, ${name}!`);
}

let k: (x: string) => void = hello; // function object
```

Los tipos lo que hacen es simplemente poner una etiqueta a las variables. Hacen que esa variable solo contenga elementos del tipo asignado, y el compilador de TS vela por que esto suceda.

Hay dos tipos de *tipos*:

- Tipos primitivos:
  - number, string, boolean.
- Tipos objetos:
  - object, array, clases..., todo lo que no es primitivo.

Concretamente hay:

- string
- number

- boolean
- null
- undefined
- symbol

## Inferencia de tipos

Los tipos no son obligatorios. Usualmente, TS los inferirá por ti. Por regla general, solo ponlo por:

- Readibilidad
- Aclaración
- Cosas que TS no pueda inferir solo
- Declarar una variable a la que asignarás un valor más tarde

TS inferirá los tipos en un array que tenga varios tipos usando el tipo que tengan todos en común. Es decir, el tipo que es compatible con todos los demás.

```
let items = [0, 1, null, 'Hi'];
```

En este caso el tipo al que todos pueden ser transformados es `string`. Además casi todos pueden ser transformados al tipo `number` así que el tipo de `items` será `(number|string)[]`.

Inferencia de tipo	Anotación de tipo
TypeScript adivina el tipo	Especificas el tipo a TypeScript

**En la práctica, procura usar la inferencia de tipo todo lo posible.**

## Aviso sobre los tipos

Para tipos como `number` y `boolean`, existe un tipo similar pero con la primera letra en mayúscula: `Number`, `Boolean`. Estos últimos hacen referencia a un objeto en desuso que no es el tipo **primitivo**, y deben ser evitados.

## Objetos

Aparte de lo usual, añadir que TS también revisa que un objeto tenga la propiedad a la que intentas acceder. Por ejemplo:

```
let employee: object;

employee = {
  firstName: 'John',
  lastName: 'Doe',
  age: 25,
  jobTitle: 'Web Developer'
};

console.log(employee.hireDate);
```

Este código nos dará el error `'hireDate' does not exist on type 'object'`.

## Tuplas

Las tuplas funcionan exactamente igual que los arrays solo que:

- El número de elementos en el array es fijo
- El tipo de los elementos es conocido
- El orden es importante

Veamos un ejemplo:

```
let skill: [string, number];
skill = ['Programming', 5]; // All good
skill = [5, 'Programming']; // Error
skill.push('No bueno'); // Error

let something: number = skill[0]; // Error, can't place a string in a number's place.
```

## Enum

`Enum` como tal no existe en JS, pero se puede hacer un objeto que emule el funcionamiento. Sin embargo en TS se puede utilizar un `enum` para hacer las cosas más simples. Véase un `enum` que representa las notas que un alumno puede tener:

```
enum Puntuation {
  Suspendido,
  Aprobado,
  Notable,
  Sobresaliente
};
```

Ahora este `enum` funciona como un tipo nuevo, el tipo `Puntuacion`. Se puede utilizar en funciones:

```
function assignNumericPuntuation(mark: Puntuacion) {
  switch(mark) {
    case Puntuacion.Suspendido:
      return 4.999;
    case Puntuacion.Aprobado:
      return 6.9;
    case Puntuacion.Notable:
      return 8.9;
    case Puntuacion.Sobresaliente:
      return 9.999;
    throw Error('Something went wrong');
  }
}

console.log(`La puntuacion de Jaime fue: ${assignNumericPuntuation(Puntuacion.Notable)}`);
```

Esto nos mostrará:

```
La puntuacion de Jaime fue: 8.9
```

Por hacerlo más legible, se sobreentiende que hay un grupo de constantes numéricas que corresponden con `Suspendido`, `Aprobado` ...

También es posible especificar los números de los miembros de un enum:

```
enum Puntuacion {
  Suspendido = 0,
  Aprobado,
  Notable,
  Sobresaliente
}
```

Esto hará que los valores siguientes a `Suspendido` sean el mismo que el anterior + 1. En este caso, `Aprobado` sería 1, y `Notable`, 2. Si se desea que los valores no sigan esta regla sería mejor especificarlos.

Como recomendación general es mejor asignarlos.

Una característica interesante de `enum` es que al ejecutar un `console.log` ...

```
enum Months {
  January = 1,
  February,
  March,
  April,
  May
}
```



```
}  
  
console.log('Months looks like:');  
for (let month in Months) {  
    console.log(month);  
}
```

Aparece lo siguiente:

```
Months looks like:  
1  
2  
3  
4  
5  
January  
February  
March  
April  
May
```

De modo que podemos acceder a los meses tanto usando el valor como el nombre del mes.

## Cuando usar un enum

- Tienes un grupo de valores fijos que tienen que ver unos con otros
- Todos esos valores son conocidos en tiempo de compilación

## Tipo any

A veces no sabes que va a ir en esa variable. Entonces puedes poner que sea de cualquier valor, `any` type. Sin embargo, no podrás llamar métodos sobre la variable que existan. Es decir, si en la variable hay un objeto, no podrás llamar a los métodos del susodicho porque es tipo `any`. Deberías meter el susodicho en una variable del tipo adecuado tras comprobar el contenido de la variable. En resumen, le informa al compilador que no compruebe el tipo.

## Tipo void

Denota la ausencia de tener un tipo. Se utiliza sobre todo para funciones que no devuelven nada. Se *puede* usar sobre variables, pero es algo ciertamente impráctico.

```
function log(message: string): void {  
    console.log(message);  
}
```

```
}
```

## Tipo never

Se utiliza para tipos que no contienen ningún valor, y no deben ser asignados a nada.

Se utiliza para:

- Funciones que lanzan un error
- Funciones en un loop infinito

```
function raiseError(message: string): never {  
    throw new Error(message);  
}
```

## Unión de tipos

Permite unir varios tipos en uno solo:

```
let result: number | string;  
result = 10; // OK  
result = 'Works'; // OK  
result = false; // One way ticket to compiling error
```

## Aliases

Se pueden renombrar algunos tipos o crear los tuyos propios:

```
type alias = existingType;  
// Algo mas realista:  
type chars = string;  
let message: chars; // Mismo efecto que poner string en lugar de chars
```

## Strings literales

A veces una función solo puede recibir una string en concreto. Para ello podemos usar strings como tipos. Hace que esa variable solo pueda ocupar esa string, y solo esa strings:

```
let click: 'click';
click = 'click'; // válido
click = 'click'; // error de compilación
```

Es muy práctico en combinación con la **unión de tipos** para reducir la cantidad de strings que puede recibir una función:

```
let mouseEvent: 'click' | 'dblclick' | 'mouseup' | 'mousedown';
mouseEvent = 'click'; // valid
mouseEvent = 'not a click'; // not valid
let anotherEvent: mouseEvent; // reusable!
```

## Funciones

Por dejar constancia, las funciones funcionan igual que el resto de cosas:

```
function echo(message: string): void {
    console.log(message.toUpperCase());
}
```

Los parámetros pueden tener tipo, y el `void` ese a la derecha de los parámetros indica que devuelve la función al terminar (nada).

## En caso de funciones flecha

```
let echo2: (message: string) => void;
echo2 = echo; // el compilador comprueba que los parámetros y valor de retorno son correctos
```

## Parámetros opcionales

Los elementos opcionales se pueden indicar en tuplas y parámetros, usando una marca `?` al final, un *postfix*. Por ejemplo:

```
function foo(a: number, b?: number) number {
    if (b) {
        return a + b;
    }
    return a + 1;
}
```

Los parámetros opcionales deben estar al final de la lista de parámetros, o TS nos dará un error.

## Parámetros 'rest'

Son los que tienen un `...` a la izquierda. Se deben anotar como un `array`.

## Clases

Por lo general, se pueden inicializar de la misma forma. Sin embargo se añaden algunas cosas interesantes.

### Modificadores de acceso

Una clase puede ocultar algunos de sus elementos al exterior haciendo uso de las etiquetas `private`, `protected` y `public`. Si no se especifica, TypeScript inferirá que es `public`.

```
class Person {
  private ssn: string;
  private firstName: string;
  private lastName: string;

  // ...
}
```

También se puede usar en funciones.

### Readonly

La propiedad `readonly` marca un atributo como inmutable, similar a `const`, de forma que hace que solo se pueda inicializar en el constructor de la clase, y en la declaración del propio elemento.

```
class Person {
  readonly birthDate: Date;

  constructor(birthDate: Date) {
    this.birthDate = birthDate;
  }
  // Also works, not recommended
  constructor(readonly birthDate: Date) {
    this.birthDate = birthDate;
  }
  //...
}
```

	readonly	const
Úsalo para	Propiedades de clases	Variables
Inicialización	Declaración/constructor de la clase	Declaración

También se puede marcar la visibilidad de elementos `private`, `public` y `protected` en el constructor, pero no se recomienda.

## Getters & Setters

TypeScript incluye estos dos también, y según [Google Style Guidelines](#) al contrario de lo que comentaron Dario y Juan García, se pueden usar siempre y cuando:

- Sean una función pura.
- Que al menos uno no sea trivial, es decir, que no sea simplemente un “pasa a través”.

```
class Foo {
  private wrappedBar = '';
  get bar() {
    return this.wrappedBar || 'bar';
  }

  set bar(wrapped: string) {
    this.wrappedBar = wrapped.trim();
  }
}

let example: Foo = new Foo();
console.log(Foo.bar); // bar
```

## Herencia

Funciona igual que en JS.

## Métodos estáticos

Funcionan igual que en JS.

## Clases abstractas

En JavaScript no existen las clases abstractas, pero en TS se pueden indicar en la definición de la clase:

```

abstract class Animal {
  name: string;
  constructor(public name_: string) {
    this.name = name_;
  }
  abstract makeSound(): void; // can also tell a function is abstract
}

class Dog extends Animal {
  constructor(name: string) {
    super(name);
  }
  makeSound() {
    console.log(`Woof!`);
  }
}

```

## Interfaces

Si nosotros quisiéramos que un objeto pasado por parámetros siguiera una anotación deberíamos hacer algo así:

```

function getFullName(person: {
  firstName: string;
  lastName: string
}) {
  return `${person.firstName} ${person.lastName}`;
}

```

Es difícil de leer, y poco práctico escribir todo eso para indicarlo.

Para ello podemos crear algo llamado una **interfaz**. Una interfaz viene a ser un objeto con ciertas propiedades que encajan en otras funciones/métodos para hacer pasar argumentos y declarar tipos de forma más cómoda.

Para declarar una interfaz, usamos la palabra reservada `interface`. Por convenio, usamos UpperCamelCase para declarar el nombre de interfaces. Como con clases.

```

interface Person {
  firstName: string;
  lastName: string;
}

```

Una vez hemos declarado la interfaz, podemos usarla como tipo:

```
function getFullName(person: Person) {
    return `${person.firstName} ${person.lastName}`;
}

let john = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(getFullName(john));
```

Nótese que la función `getFullName()` aceptará cualquier argumento que tenga al menos dos strings como propiedades. Se le pueden poner propiedades opcionales:

```
interface Person {
    firstName: string;
    middleName?: string;
    lastName: string;
}
```

Y también `readonly`.

## Tipos de funciones

Además, podemos usar las interfaces para declarar como es el tipo de una función en su declaración:

```
interface StringFormat {
    (str: string, isUpper: boolean): string
}

let format: StringFormat; // here!

format = function (str: string, isUpper: boolean) {
    return isUpper ? str.toLocaleUpperCase() : str.toLocaleLowerCase();
};

console.log(format('hi', true)); // HI
```

## Tipos de clases

Si has trabajado con Java o C#, sabrás que el principal uso de una interfaz es el de formar un contrato de uso entre clases que no tienen que ver. Es decir, especifican a clases en que formato estarán funciones o cualquier cosa que necesitemos. Por ejemplo:

```
interface Json {  
    toJSON(): string  
}
```

Esa interfaz nos especifica que la función `toJSON()` devolverá una `string` y no toma argumentos.

Podemos decir que una clase **implementa** una interfaz:

```
class Person implements Json {  
    constructor(private firstName: string,  
                private lastName: string) {  
    }  
    toJson(): string { // here!  
        return JSON.stringify(this);  
    }  
}
```

Para ello, usamos la palabra clave `implements`.

## Extender interfaces

Las interfaces, como las clases, se pueden heredar/extender. Simplemente usa la palabra clave `extend` cuando declares la interfaz:

```
interface Mailable {  
    send(email: string): boolean  
    queue(email: string): boolean  
}  
  
interface FutureMailable extends Mailable {  
    later(email: string, after: number): boolean  
}
```

En este ejemplo, `FutureMailable` incluye las funciones `send()` y `queue()`, de la interfaz `Mailable`, y además añade `later()`. Las extensiones se pueden encadenar libremente.

## Interfaces extendiendo clases

Simplemente, hace que la **interfaz solo pueda ser usada por la clase, o sus hijos**. Cualquier clase que quiera utilizar la interfaz que ha heredado de una clase también deberán heredar esa clase.

```
class Control {  
    private state: boolean;  
}
```



```
interface StatefulControl extends Control {
    enable(): void
}

class Button extends Control implements StatefulControl {
    enable() { }
}

// Error: cannot implement. Doesn't extend Control
class Chart implements StatefulControl {
    enable() { }
}
```

## Intersección de tipos

Se puede hacer intersección sobre tipos, de forma que el resultado contiene todo lo que contienen todos los tipos sobre los que hicimos la intersección. Esto se puede hacer usando `&`:

```
type typeAB = typeA & typeB;
```

Algo interesante de esto es que podemos hacer intersección de interfaces para que contengan más cosas:

```
interface BusinessPartner {
    name: string;
    credit: number;
}

interface Identity {
    id: number;
    name: string;
}

interface Contact {
    email: string;
    phone: string;
}

type Employee = Identity & Contact; // id, name, email, phone
type Customer = BusinessPartner & Contact; // name, credit, email, phone
```

Si unimos dos interfaces que tienen propiedades llamadas igual, se unirán los atributos:

```
type BusinessMan = Identity & BusinessPartner; // name, id, credit
```



Si las propiedades tienen tipos distintos se lanzará un error.

El orden no altera el resultado:

```
type typeAB = typeA & typeB;  
type typeBA = typeB & typeA;
```

## Guardias

Una guardia de tipo ayuda a TS a hacer más fácilmente los condicionales. Son básicamente lo que ya hemos visto con un añadido para las funciones si tenemos interfaces ya creadas. Haciendo referencia a las interfaces del apartado anterior:

```
function isContact(arg: any): arg is Contact { // here!  
    return partner instanceof Contact;  
}
```

## Casteo de tipos

Podemos castear de un tipo a otro (si es posible) usando la palabra reservada `as` o `<>`:

```
let a: typeA;  
let c = <typeC>a; // fine  
let b = a as typeB; // better
```

Los `<>` **no** son preferibles con respecto a `as`, debido a que:

- Son menos legibles
- No son compatibles con algunas librerías como `React`.

## Aserción de tipos

La aserción de tipos es decir al compilador en qué formato esperas tener el valor de retorno de una función que puede retornar más de un tipo:

```
function getNetPrice(price: number, discount: number, format: boolean): number | string {  
    let netPrice = price * (1 - discount);
```

```

    return format ? `${netPrice}` : netPrice;
}

let netPrice = getNetPrice(100, 0.05, true) as string;
let netPrice2 = <number>getNetPrice(100, 0.05, false);

```

## Plantillas

En inglés se llaman “Generics”. Básicamente, nos permite que una función funcione igual sin importar que le pasemos. Para ello, se usa la siguiente sintaxis:

```

function getRandomElement<T>(items: T[]): T {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}

```

La magia sucede entre los parámetros y el nombre de función: `<T>`, indica que sea lo que sea, puede aceptarlo. Similar a generic, pero especifica que si entra un argumento de tipo `T`, es retornado un elemento de tipo `T`.

Por convenio, se utiliza `T` como nombre de variable de plantilla.

## Plantillas con varios tipos

Si una función utiliza varios tipos distintos, solo tenemos que listarlos como T. Un ejemplo:

```

function merge<U, V>(obj1: U, obj2: V) {
    return {
        ...obj1,
        ...obj2
    };
}

```

## Beneficios a usar plantillas

- Puede comprobar tipos en tiempo de compilación.
- No precisa de casteos.
- Permite implementar algoritmos genéricos.

## Restricciones de plantillas

Si usamos esto tal cual, puede ser que queramos tener solo algunos tipos de objetos, no todos. Si es el caso, podemos hacer lo siguiente.

Supongamos que en la función del apartado anterior queremos que solo se le puedan pasar objetos. Para ello:

```
function merge<U extends object, V extends object>(obj1: U, obj2: V) {  
    return {  
        ...obj1,  
        ...obj2  
    };  
}
```

Al hacer que U extienda de objeto, obligamos a que U sea un objeto al pasarlo por argumentos.

También se puede usar con tipos como tal. Supongamos una función que retorna un elemento de un array dado el array y el índice:

```
function retrieve<T, K extends number>(array: T[], key: K) T {  
    return array[key];  
}
```

## Clases plantilla

Funcionan igual que las funciones.

```
class className<T>{  
    //...  
}  
  
class className<T extends TypeA>{ // constrained  
    //...  
}
```

## Interfaces plantilla

Funcionan como las clases.

```
interface interfaceName<T> {  
    // ...  
}  
  
interface Pair<K, V> {  
    key: K;
```

```
    value: V;
}
```

## Módulos en TypeScript

Funcionan exactamente igual que en JS ES6. No puedes usar `require`, solo la pareja `export` - `import`.

```
// Validator.ts
export interface Validator {
    isValid(s: string): boolean
}

// EmailValidator.ts
class EmailValidator implements Validator {
    isValid(s: string): boolean {
        const emailRegex = /^[^\\s@]+@[^\\s@]+\\.\\.[^\\s@]+$/;
        return emailRegex.test(s);
    }
}

export { EmailValidator };

// App.ts
import { EmailValidator } from './EmailValidator';

let email = 'john.doe@typescripttutorial.net';
let validator = new EmailValidator();
let result = validator.isValid(email);

console.log(result); // True
```

Se puede hacer lo mismo con los tipos, lo cual es útil para mantener una librería de tipos apartada del código base.

## Automatizar el desarrollo

<https://www.typescripttutorial.net/typescript-tutorial/nodejs-typescript/>