

TypeScript

TypeScript es un lenguaje de programación de código abierto desarrollado por Microsoft y que se basa en JavaScript. TypeScript añade conceptos comunes como clases, módulos, interfaces, genéricos y (opcionalmente) tipado estático a JavaScript. TypeScript es **fuertemente tipado** ya que requiere de que se le especifiquen los tipos de datos que se quieren utilizar.

TypeScript es un *superset* de JavaScript: todo el código JavaScript es código válido en TypeScript de manera que se puede integrar fácilmente a cualquier proyecto.

El compilador TypeScript "*transpila*" código escrito en TypeScript en código JavaScript válido y entendible por cualquier navegador que soporte Javascript.

Instalación

La forma más rápida y cómoda de instalar TypeScript es via **NPM**. TypeScript está disponible como **paquete** en el registro de [NPM](#).

Para instalar el compilador de TypeScript de forma **global** se ejecuta por consola:

```
npm install -g typescript
```

Otra forma es instalar TypeScript de forma **local** al proyecto:

```
// Inicializar un proyecto con npm (creará el fichero 'package.json')
npm init

// Instalar TypeScript de forma local
npm install --save-dev typescript
```

Podemos comprobar la versión de Typescript instalada:

```
tsc -v
```

Usando TypeScript

Para compilar un fichero TypeScript llamado `main.ts` escribimos en el terminal:

```
tsc main.ts
```

Para no tener que compilar un fichero TypeScript cada vez que se realicen cambios, podemos arrancar el compilador TypeScript en modo '*watch*' de forma que compilará el fichero TypeScript cada vez que detecte un cambio. :

```
// Se finaliza el proceso con 'Ctrl + C'
tsc main.ts -w
```

Para inicializar un proyecto TypeScript, escribimos por terminal dentro de la carpeta del proyecto:

```
tsc --init
```

Esto crea un fichero `tsconfig.json` con las opciones por defecto. La presencia de este archivo significa que este directorio es la raíz del proyecto. Un ejemplo de este fichero sería:

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

Algunas opciones son:

- `target` : la versión de JavaScript a la que se tiene que compilar.
- `module` : el sistema de módulos a utilizar
- `strict` : habilitar/deshabilitar todas las opciones estrictas de comprobación de tipos
- `outDir` : el directorio de salida de los ficheros JavaScript compilados
- `rootDir` : la carpeta raíz donde se ubican los ficheros TypeScript
- `noImplicitAny` : habilitar/deshabilitar la generación de informes de error para expresiones y declaraciones con un tipo 'any' implícito
- `sourceMap` : genera ficheros `*.map` en la compilación de ficheros

[Más información sobre este fichero 'tsconfig.json'](#)

Es importante indicar que cuando se utiliza un fichero `tsconfig.json` con las opciones de compilación, no se debe indicar el nombre del fichero o los ficheros con el código Typescript, ya que compilará todos los ficheros `.ts` del proyecto.

Para depurar el código TypeScript en el navegador, debemos utilizar un fichero `*.map` de forma que el navegador pueda relacionar el código Javascript que está ejecutando con el código fuente escrito en TypeScript. Este fichero se genera indicando `"sourceMap": true` en el fichero de configuración `tsconfig.json`.

Disponemos de varios editores online o *playground* para escribir y probar código escrito en TypeScript como puede ser el [editor oficial](#).

Visual Studio Code es un editor que incluye soporte para TypeScript aunque no incluye el compilador `tsc`. [Más información sobre VSCode y TypeScript](#).

Tipos básicos

Para que los programas sean útiles, debemos poder trabajar con algunas de las unidades de datos más simples: números, cadenas, estructuras, valores booleanos y similares. En [TypeScript](#), se admite la mayoría de los tipos que se esperaría en JavaScript.

El tipo de la variable se indica después del nombre. Se separa el nombre de la variable y el tipo mediante dos puntos ':', como por ejemplo:

```
let isDone: boolean = false;
```

Cuando una variable se define de un tipo, no se pueden asignar valores de otro tipo a esa variable. Si se intenta asignar otro tipo se obtiene un error en tiempo de compilación:

```
let isVisible: boolean = true;
isVisible = "hidden"; // Error: string not assignable to boolean
```

Cuando se asigna un valor a una variable, TypeScript puede **inferir el tipo** de la variable si no se indica explícitamente. Por tanto, se puede omitir la anotación de tipo cuando se declara una variable y se asigna un valor en la misma línea:

```
let isVisible = true; // TypeScript infers the type boolean
```

Cuando se separa la declaración y la inicialización en varias líneas, el compilador de TypeScript infiere el tipo `any` lo que significa que se podrá asignar cualquier valor a esa variable:

```
let isVisible; // declaration without type annotation
isVisible = true; // assignment of bool
isVisible = "hidden"; // WORKS!!! No compile-time error
```

Por tanto, para asegurarnos que TypeScript pueda realizar la comprobación de tipos y así evitar errores, si la declaración y la inicialización se producen en líneas diferentes, es recomendable definir el tipo en la declaración:

```
let isVisible: boolean; // declaration with type annotation
isVisible = true; // assignment of bool
isVisible = "hidden"; // Error: string not assignable to boolean
```

Boolean

El tipo de datos más básico es el tipo `boolean` que admite los valores 'true/false':

```
let isDone: boolean = false;
```

Number

Dado que TypeScript es un superconjunto de JavaScript, todos los números en TypeScript son números de 64bits en punto flotante. Estos números de punto flotante obtienen el tipo `number`. Además de los literales hexadecimales y decimales, TypeScript también admite literales binarios y octales introducidos en la especificación ES2015.

```
let width: number = 2;
let decimal: number = 6.5;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

String

Para crear una variable de tipo cadena, usamos el tipo `string`, presente en infinidad de lenguajes. Al igual que JavaScript, TypeScript permite el uso de comillas dobles (") o comillas simples (') para rodear cadenas.

```
let color: string = "blue";
color = "red";
let name = "John";
let firstName = "Doe";
```

TypeScript tiene soporte para los *'template strings'* como en Kotlin, que pueden abarcar varias líneas y tener expresiones incrustadas. Estas cadenas están rodeadas por el carácter de comillas invertidas o *'backquote'* (```), y las expresiones incrustadas tienen la forma `${expr}`:

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;

// Cadena de tipo 'string' de varias líneas y con una expresión incrustada.
let sentence: string = `Hello, my name is ${fullName}.

I'll be ${age + 1} years old next month.`;
```

La forma anterior es equivalente a declarar la variable `sentence` de la siguiente forma, que es la forma clásica de concatenación y que en última instancia es la forma que se genera cuando el compilador compila el código TypeScript en código Javascript (dependiendo del *target*):

```
let sentence: string = "Hello, my name is " + fullName + ".\n\n" +
    "I'll be " + (age + 1) + " years old next month.";
```

Cuando se compila con `"target": "es5"` o posterior el fichero `tsconfig.json`, los *'template strings'* están soportados nativamente, por lo que el código Javascript compilado será el mismo que el código Typescript.

Array

TypeScript, como JavaScript, permite trabajar con arrays de valores. Los arrays se pueden escribir de dos maneras. Una forma es indicar el tipo de los elementos seguidos de `[]` para indicar que es un array de elementos de ese tipo:

```
// Declaración e inicialización de un array
let list: number[] = [1, 2, 3];
let firstNames: string[] = ["Thomas", "Sara", "Julia"];
```

La otra forma es usar el tipo de array genérico, `Array<elemType>`:

```
let list: Array<number> = [1, 2, 3];
```

Podemos añadir valores a un array mediante `push()` o mediante asignación directa por posición:

```
const myArray: number[] = [];

myArray.push(1);
myArray.push(2);
```

```
myArray.push(3);  
myArray[4] = 4;
```

Iterar por un array con `for...of` and `for...in`

Para iterar por los valores de un array podemos usar un bucle `for-of` :

```
let firstnames: string[] = ["Julia", "Anna", "Thomas"];  
for (let firstname of firstnames) {  
    // Mostrará en la consola del navegador el contenido del array  
    console.log(firstname);  
}
```

En TypeScript existen además los bucles `for-in` que en vez de devolver el contenido del array devuelve el índice:

```
let firstnames: string[] = ["Julia", "Anna", "Thomas"];  
for (let index in firstnames) {  
    console.log(`${index} - ${firstnames[index]}`);  
}  
// This code prints:  
// 0 - Julia  
// 1 - Anna  
// 2 - Thomas
```

Tupla

Las tuplas permiten expresar un array con un número fijo de elementos cuyos tipos son conocidos, aunque no necesariamente iguales. Por ejemplo, podemos usar una tupla para representar un valor que se compone de un `string` y un `number`, de forma que el `string` está en el índice 0 y el `number` está en el índice 1. El compilador conoce esto y puede realizar las comprobaciones al asignar nuevos valores:

```
// Declarar una tupla  
let x: [string, number];  
// Inicializar la tupla  
x = ["hello", 10]; // OK  
// Inicializar la tupla con valores de tipo incorrecto  
x = [10, "hello"]; // Error  
  
// Declarar e inicializar una tupla en la misma línea  
let nameIsDev: [string, boolean] = ["Thomas", true];
```

Al acceder a un elemento con un índice conocido, se recupera el tipo correcto:

```
console.log(x[0].substring(1)); // OK  
console.log(x[1].substring(1)); // Error, 'number' does not have 'substring'
```

El acceso a un elemento fuera del conjunto de índices conocidos falla con un error:

```
x[3] = "world"; // Error, Property '3' does not exist on type '[string, number]'.  
  
console.log(x[5].toString()); // Error, Property '5' does not exist on type '[string, number]'.
```

Enum

Un añadido útil al conjunto estándar de tipos de datos de JavaScript es la enumeración. Al igual que en lenguajes como C# o Java, una enumeración es una forma de dar nombres más amigables a conjuntos de valores numéricos. Para acceder al valor de la enumeración, usamos su nombre seguido de un punto y el nombre de la variable miembro como por ejemplo `Color.Green` o `Color.Blue` :

```
enum Color {  
  Red,  
  Green,  
  Blue  
}  
let c: Color = Color.Green;
```

De forma predeterminada, las enumeraciones comienzan a numerar sus miembros a partir de 0. Puede cambiar esto configurando manualmente el valor de uno de sus miembros. Por ejemplo, podemos comenzar el ejemplo anterior en 1 en lugar de 0:

```
enum Color {Red = 1, Green, Blue}  
let c: Color = Color.Green;
```

O bien, establecer manualmente todos los valores en la enumeración:

```
enum Color {Red = 1, Green = 2, Blue = 4};  
let c: Color = Color.Green;
```

Una característica útil de las enumeraciones es que podemos recuperar el nombre utilizando el valor de la enumeración o recuperar el valor usando el nombre:

```
enum Color {Red = 1, Green, Blue}  
let colorName: string = Color[2]; // Usamos el valor para recuperar el nombre  
let valueColor: number = Color["Green"];  
  
console.log(colorName); // Se muestra 'Green' que es nombre con valor 2  
console.log(valueColor); // Se muestra '2' que es el valor de 'Green'
```

Any

En determinados escenarios es posible que tengamos que describir una variable con un tipo que es desconocido dado que su valor puede provenir de contenido dinámico, como por ejemplo, del usuario o de una biblioteca de terceros.

En estos casos, podemos optar por indicar al compilador de TypeScript que no realice la verificación de tipos ni la existencia de sus miembros o métodos. Para ello, usamos el tipo `any` :

```
let notSure: any = 4;  
notSure = "maybe a string instead";  
console.log(notSure.length); //Imprime 22  
  
notSure = false; // okay, definitely a boolean  
  
/* El compilador no emite ningún error e imprime 'undefined' */
```

```
dado que el tipo boolean no tiene el atributo 'length' */
console.log(notSure.length);
```

El tipo `any` es una forma poderosa de trabajar con código JavaScript existente, permitiéndole optar gradualmente por la verificación de tipos durante la compilación.

Además de las variables, el tipo `any` es especialmente importante en los parámetros de las funciones. Si no se especifica el tipo, los parámetros son implícitamente de tipo `any`:

```
// El parámetro 'friend' es de tipo 'any'
function printFirstName(friend) {
  console.log(friend.firstName);
}
```

Cuando estamos migrando código Javascript legado, podemos indicar al compilador que nos marque como error (y que sea visible en el editor) si se lo indicamos con `"noImplicitAny": true` en el fichero `tsconfig.json`. De esta forma, para solucionar el error deberemos indicar de forma explícita el tipo `any` en el parámetro de la función. No es que el error se deba al tipo, ya que el tipo `any` es un tipo válido si no que se debe a que se debe indicar de forma explícita:

```
// Ahora el parámetro 'friend' es de tipo 'any' de forma explícita
function printFirstName(friend: any) {
  console.log(friend.firstName);
}
```

El tipo `any` también es útil si conoce alguna parte del tipo, pero tal vez no toda. Por ejemplo, puede tener un array pero el array tiene una mezcla de diferentes tipos, de forma que si indicamos `any` permitimos al array almacenar cualquier tipo:

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

Para los casos en los que se tiene la información en tiempo de compilación, siempre es recomendable indicar el tipo de forma explícita en vez de emplear el tipo `any`, ya sea de forma explícita o implícita, ya que esto permitirá al compilador de TypeScript realizar la verificación de tipos y el soporte de herramientas como la finalización de declaraciones.

Union Types

En determinados escenarios podemos necesitar almacenar diferentes tipos en una misma variable pero queremos mantener acotado los tipos posibles. En vez de usar el tipo `any` que permitiría cualquier tipo podemos usar el *'union type'*. Este tipo es una combinación de los tipos posibles que admitirá la variable.

Por ejemplo, definimos una variable usando el *'union type'* `boolean|number` de forma que la variable sólo acepta valores de tipo `boolean` o `number`. Cualquier intento de asignar valores de otro tipo lanza un error:

```
let isVisible: boolean | number = true;
/* El compilador genera un error en tiempo de compilación
ya que el tipo 'boolean' no tiene la propiedad 'length' */
console.log(isVisible.length);

isVisible = "Yes, is visible";
/* Ahora el compilador no genera ningún error ya que
ahora la variable almacena un 'string' que sí tiene la propiedad 'length' */
console.log(isVisible.length);
```

Void

El tipo `void` es la ausencia de tener un tipo. Normalmente se utiliza como tipo de retorno de funciones que no devuelven un valor:

```
function warnUser(): void {
  console.log("This is my warning message");
}
```

El tipo `void` no es necesario indicarlo dado que el compilador infiere el tipo de retorno:

```
function warnUser() {
  console.log("This is my warning message");
}
```

Declarar variables de tipo `void` no es útil porque solo puedes asignarles `undefined` o `null`:

```
let unusable: void = undefined;
```

'Null' and 'Undefined'

Indefinido y nulo son valores que en JavaScript conducen a muchos errores. En TypeScript, tanto los valores indefinidos como los valores nulos en realidad tienen sus propios tipos llamados `undefined` y `null` respectivamente. Al igual que `void`, no son extremadamente útiles por sí solos:

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

Por defecto, `null` y `undefined` son subtipos de todos los demás tipos. Eso significa que puede asignar `null` o `undefined` a algo como `number` o `string`:

```
let firstName: string = "Thomas";
firstName = null; // OK
firstName = undefined; // OK
```

Sin embargo, cuando se usa el indicador `--strictNullChecks` o `"strictNullChecks": true` en el fichero `tsconfig.json`, los tipos `null` y `undefined` ya no se comportan como subtipos de todos los demás tipos y sólo se pueden asignar a una variable de uno de sus tipos respectivos (la única excepción es que `undefined` también se puede asignar a `void`).

Never

El tipo `never` representa el tipo de valores que nunca ocurren. Por ejemplo, `never` es el tipo de retorno para una expresión de función o una expresión de función de flecha que siempre arroja una excepción o una que nunca devuelve un valor.

El tipo `never` es un subtipo de cada tipo y por tanto es asignable a todos los demás tipos. Sin embargo, ningún tipo es un subtipo de `never` ni asignable a `never` excepto sí mismo. Incluso `any` no es asignable a `never`.


```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {}
}
```

Object

`Object` es un tipo que representa el tipo no primitivo, es decir, cualquier cosa que no sea `number`, `string`, `boolean`, `symbol`, `null`, o `undefined`.

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

El tipo `object` se puede usar para representar datos en estructuras más complejas. Puede contener propiedades en forma de variables que pueden ser de tipo `string`, `number`, arrays, funciones y otros objetos:

```
const myObject = {
    title: 'Example Object',
    value: 5
}

console.log(myObject.value);
console.dir(myObject)
```

Aserciones de tipo

Una aserción de tipo es como una conversión de tipo en otros lenguajes, pero no realiza ninguna verificación especial o reestructuración de datos. Se utiliza la palabra clave `as`:

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

No tiene impacto en el tiempo de ejecución, y es utilizado exclusivamente por el compilador. TypeScript asume que se han realizado las comprobaciones necesarias. Además, al utilizar la aserción de tipo nos podemos valer del autocompletado de cualquier editor:

```
let someValue: any = "this is a string";

// Pese a escribir mal 'length', el compilador no muestra ningún error
let strLength: number = someValue.lengt;

/* Ahora, si escribimos mal 'length' el compilador nos mostrará un error.
Además, el autocompletado del editor autocompleta correctamente el nombre de la variable 'length' */
let strLength: number = (someValue as string).length;
```

Declaración de variables

Las variables en JavaScript siempre se han declarado con la palabra clave `var`. En la especificación ES2015 se introdujeron las nuevas palabras clave `let` y `const`, que por supuesto también están disponibles en TypeScript. Se recomienda usar `let` y `const` en lugar de `var`.

var VS let

`let` permite declarar variables limitando su alcance ('scope') al bloque, declaración, o expresión donde se está usando mientras que con `var` se define una variable global o local en una función sin importar el ámbito del bloque.

La declaración de variables con `let` y `const` es similar a `var`. Tan sólo hay que cambiar la palabra clave:

```
var firstName: string = "John";

let lastName: string = "Doe";
const nonChangeableName: string = "Julia";
```

La diferencia radica en el alcance de cada variable según como su definición. Es recomendable usar `let` o `const` en vez de `var`.

Alcance de bloque

Las variables declaradas con `var` tienen un alcance de función o '*function-scoped*' mientras que las variables declaradas con `let` tienen un alcance de bloque o '*block-scoped*' que es más parecido a Java.

En el siguiente ejemplo tenemos una variable declarada con `var` dentro del ámbito de un `if`, lo que se traduce en que la variable es accesible desde cualquier punto de la función, tanto dentro como fuera del ámbito del `if` donde fue declarada:

```
function getNumber(init) {
  if (init) {
    var x = 9;
  }
  return x; // Correcto
}
```

Si ahora cambiamos la declaración por `let`, sólo podremos acceder a la variable dentro del `if` ya que ahora su alcance es de bloque:

```
function getNumber(init) {
  if (init) {
    let x = 9;
  }
}
```

```
    return x; // Error: x not visible here, as "let" is block-scoped
}
```

Otra propiedad de las variables de ámbito de bloque es que no se pueden leer ni escribir antes del punto donde se declaran:

```
a++; // illegal to use 'a' before it's declared;
let a;
```

Declaraciones múltiples

Una variable declarada con `var` se puede declarar varias veces. El código resultante es perfectamente válido:

```
var firstName: string = "John";
var firstName: string = "Julia";
```

En cambio, si declaramos las variables con `let`, no se pueden declarar varias variables con el mismo nombre dentro del mismo bloque. Si se hace el compilador genera un error en tiempo de compilación:

```
let firstName: string = "Thomas"; // Compile-time error
let firstName: string = "Julia"; // Compile-time error
```

En cambio sí que podemos declarar variables con el mismo nombre usando `let` pero en diferentes bloques. Este concepto se llama *'shadowing'* y significa que el bloque más interior *'oculta'* la variable más exterior. Aún teniendo el mismo nombre, son variables diferentes y que pueden tener valores diferentes. Por tanto, según en el bloque que nos encontremos tendremos acceso a una u otra variable:

```
let firstName: string = "Thomas";
{
    let firstName: string = "Bill";
    console.log(firstName); // Logs "Bill"
}
console.log(firstName); // Logs "Thomas"
```

Para evitar confusiones, es recomendable usar diferentes nombres para las variables en vez de usar el concepto de *'shadowing'*, que también está presente en lenguajes como Java.

Declarar variables antes de su uso

Otra regla de `let` es que la variable debe ser declarada antes de su uso. El siguiente ejemplo lanza un error:

```
console.log(firstName); // Error: firstName used before declaration
let firstName: string = "Thomas";
```

No es importante el número de línea de la declaración o lugar de la declaración, sino el flujo de cómo se ejecuta su código. En el siguiente código, aunque el uso de la variable aparece antes, realmente se está usando después de su declaración:

```
function log() {
    console.log(firstName); // Ok to access firstName here
}
```

```
let firstName: string = "Thomas";
log(); // Ahora se usará realmente la variable, después de su declaración y por tanto el código es correcto
```

Uso de `const`

Todas las reglas aplicables a `let` se aplican también a `const`. Las variables declaradas con `const` son también variables con un alcance de bloque o *'block-scoped'*. La única diferencia con `let` es que la variable declarada con `const` sólo se le puede asignar un valor una sola vez y esta asignación se debe hacer en la declaración de la variable. Por tanto, si tenemos una variable cuyo valor no va a cambiar en el tiempo debemos declararla como `const`:

```
const firstName: string = "John";
firstName = "Julia"; // Error: firstName is a const
```

Cuando se asigna un objeto a una variable declarada como `const` se pueden cambiar las propiedades a posteriori pero no se podrá reasignar otro objeto diferente a la variable:

```
const friend = { firstName: "Thomas", lastName: "TypeScript" };
friend.firstName = "Julia"; // OK
friend.lastName = "Huber"; // OK
friend = { firstName: "x", lastName: "y" }; // Error: friend is const
```

Control de flujo

Estructuras condicionales

```
// 'if' statement
const age = 21;

if (age > 40) {
  // Code to execute if age is greater than 40
} else if (age > 18) {
  // Code to execute if age is greater than 18
  // but less than 41
} else {
  // Code to execute in all other cases
}

// 'switch'
const styles = {
  tranditional: 1,
  modern: 2,
  postModern: 3,
  futuristic: 4
};
const style = styles.tranditional;
switch (style) {
  case styles.tranditional:
    // Code to execute for traditional style
    break;
  case styles.modern:
    // Code to execute for modern style
    break;
  case styles.postModern:
    // Code to execute for post modern style
    break;
  case styles.futuristic:
    // Code to execute for futuristic style
    break;
}
```

```
default:
  throw new Error('Style not known: ' + style);
}
```

Estructuras de tipo bucle

```
// 'for' Loop
const names = ['Lily', 'Rebecca', 'Debbie', 'Ann'];

for (let i = 0; i < names.length; i++) {
  console.log(names[i]);
}

for (let name in names) {
  console.log(name);
}

// 'while' Loop
let counter = 10;

while (counter > 0) {
  counter--;
  console.log(counter);
}

// 'do-while' Loop
do {
  counter--;
  console.log(counter);
} while (counter > 0);
```

Interfaces y clases

En TypeScript se pueden emplear construcciones orientadas a objetos como interfaces, clases y herencia. Las clases forman parte de la especificación ES2015 pero las interfaces siguen siendo un concepto disponible sólo en TypeScript.

Interfaces

Una [interfaz](#) es una construcción TypeScript. No hay salida compilada en JavaScript. Esto se debe a que una interfaz es solo un tipo sin implementación. Como JavaScript no tiene tipos y no hay implementación para una interfaz, no hay nada que generar en JavaScript.

Uno de los principios básicos de TypeScript es la verificación de tipos. Dado que una interfaz es solo un tipo sin implementación, es una forma poderosa de crear contratos de forma que si no se satisface la interfaz, TypeScript mostrará un error en tiempo de compilación.

Como ejemplo, cuando una función define un parámetro sin un tipo explícito, el compilador le asigna el tipo `any` como ya vimos, con lo cual el compilador no puede realizar ninguna verificación de tipos y nuestro código es propenso a errores:

```
// Función con el parámetro 'friend' de tipo 'any'
function getFullName(friend) {
  let fullName = friend.firstName;
  if (friend.lastName) {
    fullName += " " + friend.lastName;
  }
  return fullName;
}
```

En cambio, con una interfaz podemos restringir el tipo pasado a la función y permitir a TypeScript realizar la comprobación de tipos.

Una interfaz tiene un nombre, puede tener ninguna, una o varias propiedades y ninguno, uno o varios métodos que las clases que implementen la interfaz deberán implementar. Puede que no todas las propiedades de una interfaz sean obligatorias. TypeScript permite las propiedades opcionales, que se indican con el símbolo '?':

```
interface Friend {
  firstName: string;
  lastName?: string; // optional
}

function getFullName(friend: Friend) {
  let fullName = friend.firstName;
  if (friend.lastName) {
    fullName += " " + friend.lastName;
  }
  return fullName;
}

console.log(getFullName({ firstName: "Thomas", lastName: "Huber" }));
console.log(getFullName({ firstName: "Thomas" })); // LastName is optional
console.log(getFullName({})); // Error: firstName is missing

console.log(getFullName(25)); // Error: Argument of type '25' is not assignable to parameter of type 'Friend'
```

Propiedades de sólo lectura

Se puede definir propiedades de sólo lectura con la palabra clave `readonly`. Una propiedad de sólo lectura debe inicializarse en su declaración:

```
interface Developer {
  readonly knowsTypeScript: boolean;
}
```

Implementar una interfaz

Las clases en TypeScript pueden implementar una o varias interfaces usando la palabra clave `implements`. Para implementar varias interfaces, se separan los nombres con comas. En el ejemplo la clase `Friend` implementa la interfaz `Person`:

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

  getFullName(): string;
}

class Friend implements Person {
  firstName: string;
  lastName?: string; //optional

  constructor(firstName: string, lastName?: string) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName(): string {
    let fullName = this.firstName;
    if (this.lastName) {
      fullName += " " + this.lastName;
    }
  }
}
```

```

    return fullName;
  }
}

let friend: Friend = new Friend("John", "Doe");
console.log(friend.getFullName()); // Prints 'John Doe'

```

Implementar una interfaz en una clase nos asegura que la clase implementa los mismos métodos y propiedades de la interfaz.

Sin embargo en TypeScript, a diferencia de Java o C#, para verificar la compatibilidad de tipos se emplea la *'structural typing'* y no la *'nominal typing'* que se emplea en Java o C#. Esto significa que los miembros del objeto son importantes y no el tipo en si mismo:

```

interface Developer {
  knowsTypeScript: boolean;
}

class Friend {
  knowsTypeScript: boolean;
}

```

En el ejemplo tenemos una interfaz con una propiedad de tipo `boolean` y una clase con la misma propiedad y el mismo tipo `boolean`. La única diferencia es que una es un interfaz y la otra es una clase pero estructuralmente son iguales ya que tienen la misma propiedad.

En TypeScript podremos usar la clase `Friend` en aquellos lugares donde se requiera un objeto de tipo `Developer` ya que ambos son estructuralmente iguales y sin que la clase `Friend` haya implementado la interfaz `Developer`:

```

let dev: Developer = new Friend(); // OK, because property exists

```

Clases

La especificación de Javascript ES2015 tiene soporte para las clases. Gracias a TypeScript, podemos usar clases y compilar el código a 'ES5' o incluso a 'ES3'.

Una `clase` tiene propiedades, métodos y un constructor usado para instanciar la clase:

```

class Friend {
  firstName: string;
  lastName?: string; //optional

  constructor(firstName: string, lastName?: string) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName(): string {
    let fullName = this.firstName;
    if (this.lastName) {
      fullName += " " + this.lastName;
    }
    return fullName;
  }
}

```

El constructor crea una instancia de la clase e inicializa las propiedades. Con la palabra clave `this` se hace referencia a la instancia actual de la clase, similar a C# o Java.

Para instanciar una clase usamos la palabra clave `new` al igual que en C# o Java. En el constructor podemos omitir los parámetros opcionales:

```
let friend1 = new Friend("Thomas", "Huber");
let friend2 = new Friend("Julia");           // se omite el parámetro opcional 'lastName'
```

Constructor

TypeScript sólo permite **un único constructor** a diferencia de C# o Java. El mecanismo para poder crear objetos de forma flexible con un único constructor es mediante los parámetros opcionales y los valores por defecto para los parámetros.

Pongamos por ejemplo una clase Java con tres constructores que admiten uno, dos y tres parámetros para construir un objeto de ese tipo, siendo sólo uno de los parámetros requerido ya que se repite en los tres constructores:

```
class Friend {
    String fullName = "";
    int age = 0;
    boolean knowsTypeScript = true;

    Friend(String fullName) {
        this.fullName = fullName;
    }

    Friend(String fullName, int age) {
        this.fullName = fullName;
        this.age = age;
    }

    Friend(String fullName, int age, boolean knowsTypeScript) {
        this.fullName = fullName;
        this.age = age;
        this.knowsTypeScript = knowsTypeScript;
    }
}

class Main {
    public static void main(String[] args) {
        Friend friend = new Friend("John Doe");
        Friend friend2 = new Friend("John Doe", 40);
        Friend friend3 = new Friend("John Doe", 40, false);
    }
}
```

Para definir la misma clase en TypeScript con un sólo constructor usamos los parámetros opcionales para indicar los que son opcionales y los valores por defecto para asignar valores a los parámetros:

```
class Friend {
    constructor(public fullName: string, public age?: number, public knowsTypeScript: boolean = true) { }
}

// Podemos crear tres objetos con distinto número de parámetros usando el mismo constructor
let friend: Friend = new Friend("John Doe");
let friend2: Friend = new Friend("John Doe", 40);
let friend3: Friend = new Friend("John Doe", 40, false);

console.log(friend.fullName); // Prints 'John Doe'
```



```
console.log(friend.age); // Prints 'undefined' ya que no le hemos asignado un valor ni tiene valor por defecto
console.log(friend.knowsTypeScript); // Prints 'true' que es el valor por defecto
```

Propiedades en los parámetros del constructor

'*Parameter properties*' es una forma directa en TypeScript de definir propiedades de forma implícita que serán definidas e inicializadas por el compilador a partir de los parámetros del constructor.

Para indicar al compilador que es un '*parameter property*' se añade el modificador de visibilidad al parámetro en el constructor. El compilador definirá e inicializará una propiedad con el mismo nombre que el parámetro de forma automática:

```
// Clase con una propiedad que se inicializa en el constructor
class Friend {
  firstName: string;
  lastName?: string; // optional

  constructor(firstName: string, lastName?: string) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}

/* Misma clase pero haciendo uso de 'parameter properties'.
Al añadir 'public' el compilador generará e inicializará la propiedad automáticamente */
class Friend {
  constructor(public firstName: string, public lastName?: string) { }
}
```

'Getters/Setters'

TypeScript admite '*getters/setters*' como una forma de interceptar accesos a un miembro de un objeto como ocurre en C#. La única limitación es que tenemos que indicar un *target* 'ES5' o superior. En la especificación 'ES3' o inferior no está soportado.

```
class Friend {
  private _firstName: string;

  set firstName(value: string) {
    this._firstName = value;
  }

  get firstName(): string {
    return this._firstName;
  }
}

let friend = new Friend();
// A diferencia de Java, La notación es 'invisible' como si no se estuviera haciendo uso del 'set()'
friend.firstName = "Thomas";
console.log(friend.firstName); // ni del 'get()'
```

Cuando no incluimos un método `set()` el compilador infiere la propiedad como de sólo lectura de forma automática, con lo que desde dentro de la clase podemos asignar valores a la propiedad pero desde fuera de la clase obtendremos un error en tiempo de compilación:

```
class Friend {
  private _firstName: string = "John";

  get firstName(): string {
    return this._firstName;
  }
}
```

```

    }
}

let friend = new Friend();
console.log(friend.firstName); // OK
friend.firstName = "Julia"; // Error, as property is readonly

```

Propiedades de sólo lectura

Se puede definir propiedades de sólo lectura con la palabra clave `readonly`. Una propiedad de sólo lectura debe inicializarse en su declaración o en el constructor. En otros lugares no se puede asignar un nuevo valor.

Una propiedad se puede definir también como sólo lectura en una *'parameter property'*.

```

class Friend {
    public readonly firstName: string;

    constructor(firstName: string) {
        this.firstName = firstName;
    }
}

class Friend {
    // Parameter property
    constructor(public readonly firstName: string) {}
}

```

Propiedades estáticas

TypeScript admite [propiedades estáticas](#). Las propiedades estáticas pertenecen a la clase y no a una instancia de la clase. Eso significa que su valor existe solo una vez, sin importar cuántas instancias se creen de la clase.

```

class Friend {
    static friendCounter: number = 0;
    constructor() {
        Friend.friendCounter++;
    }
}

new Friend();
new Friend();
new Friend();
console.log(Friend.friendCounter); // Logs 3

```

Modificadores de acceso

Por defecto, todos los miembros de una clase como propiedades, métodos y el constructor son **públicos** en [TypeScript](#). El modificador `public` significa que se puede acceder a cualquier miembro de una clase que sea público desde fuera de esa clase. Se puede marcar como `public` de forma explícita, pero no es necesario salvo por legibilidad.

Los modificadores de acceso son muy parecidos a Java:

- `public` -> miembro visible desde fuera de la clase
- `private` -> miembro sólo visible dentro de la clase y no fuera de ella.
- `protected` -> miembro sólo visible dentro de la clase y en las subclases, pero no fuera de ella.

Herencia

TypeScript tiene soporte para la [herencia](#), uno de los pilares de la **"Programación Orientada a Objetos"**. Al igual que Java se usa la palabra reservada `extends` para heredar de una clase existente:

```
class Friend {
  constructor(public firstName: string) {}
}

class Developer extends Friend {
  knowsTypeScript: boolean;
}
```

La subclase tiene sus propiedades y las propiedades heredadas de la clase padre.

Para instanciar una subclase se utiliza el constructor de la clase padre o su propio constructor si lo tiene definido. La única regla es que el constructor de la subclase **debe llamar al constructor de la superclase**. Para ello se utiliza `super()` :

```
class Friend {
  constructor(public firstName: string) {}
}

class Developer extends Friend {
  // 'firstName' es una propiedad normal que se utilizará para llamar al constructor de la superclase
  // En cambio 'knowsTypeScript' es una 'parameter property'
  constructor(firstName: string, public knowsTypeScript: boolean) {
    super(firstName);
  }
}
```

Clases abstractas

TypeScript tiene soporte para el concepto de [clases abstractas](#). Con una clase abstracta se fuerza a la subclase a que implemente los métodos abstractos.

Para crear una clase abstracta se utiliza la palabra reservada `abstract` . Los métodos abstractos que tienen que ser implementados también se marcan con la palabra `abstract` .

```
abstract class Friend {
  constructor(public firstName: string) {}
  abstract sayHello(): void;
}

class Developer extends Friend {
  knowsTypeScript: boolean = true;
  sayHello() {
    console.log(`Hi, I'm ${this.firstName}`)
  }
}

let dev: Developer = new Developer("John");
dev.sayHello(); // Prints 'Hi, I'm John'
```

El operador 'instanceof'

Para saber si un objeto es de un tipo utilizamos el operador `instanceof` . Al igual que Kotlin y su *'smart cast'* una vez hemos hecho la comprobación de tipo en un `if` , podemos usar el objeto dentro del bloque sin realizar una aserción de tipo, es decir, sin utilizar `as` :

```

class Friend {
    constructor(public firstName: string) {}
}

class Developer extends Friend {
    knowsTypeScript: boolean;
}

class ExcelGuru extends Friend {}

// La variable recibe un objeto y no sabemos si es un 'Developer' o un 'ExcelGuru'
let friend: Friend = methodReturnsOneDeveloperOrExcelGuru();

if(friend instanceof Developer) {
    console.log("Yeah, it's a dev");
    console.log("Knows TypeScript: " + friend.knowsTypeScript)

    // No es necesario usar una aserción de tipo
    // console.log("Knows TypeScript: " + (friend as Developer).knowsTypeScript)
}

```

Desestructurando objetos

Al igual que Kotlin, TypeScript permite desestructurar objetos, lo que significa que podemos extraer una o más propiedades de un objeto con una notación más compacta. Podemos asignar los valores de las propiedades a variables nuevas o utilizar los mismos nombres de las propiedades como nombres de variable:

```

class Friend {
    constructor(public firstName: string, public lastName: string, public isDeveloper: boolean) { }
}

let friend = new Friend("John", "Doe", true);

// Forma clásica
let surname = friend.lastName;
let isDev = friend.isDeveloper;

// Desestructurar un objeto en Las nuevas variables 'surname' y 'isDev' de forma compacta
let {lastName: surname, isDeveloper: isDev} = friend;
console.log(surname);
console.log(isDev);

// Podemos desestructurar el objeto haciendo uso de Los mismos nombres que Las propiedades
let {lastName, isDeveloper} = friend;
console.log(lastName);
console.log(isDeveloper);

```

Esta notación más compacta también se aplica a objetos cuando son retornados por una función:

```

function loadFriend(): Friend {
    return new Friend("Julia", "Huber", false);
}

let {firstName} = loadFriend();
console.log(firstName);

```

También se puede utilizar con arrays:

```

let numbers: number[] = [1,2,3,4];

```

```
let [first, second] = numbers;
console.log(first); // Prints '1'
console.log(second); // Prints '2'
```

Funciones

Las **funciones** son la base fundamental de cualquier aplicación en JavaScript. En TypeScript, aunque hay clases, espacios de nombres y módulos, las funciones siguen desempeñando un papel clave en la descripción de cómo hacer las cosas. TypeScript también agrega algunas capacidades nuevas a las funciones estándar de JavaScript para que sea más fácil trabajar con ellas.

Tipos de funciones

JavaScript soporta dos tipos de funciones:

- Funciones con nombre o *'named functions'*:

```
// 'Named function'
function multiply(x, y) {
  return x * y;
}
```

- Funciones anónimas o *'anonymous functions'*:

```
// Anonymous function
let add = function(x, y) { return x + y; };
```

Las funciones anónimas no tienen un nombre que permita hacer referencia a la función y así poder invocarla pero pueden ser asignadas a una variable. Una vez asignada a una variable, puede ser llamada igual que una función con nombre:

```
let resultMul = multiply(3, 3); // Función con nombre
let resultAdd = add(3, 3); // Función anónima asignada a la variable 'add'
```

Con TypeScript podemos indicar de forma explícita el tipo de los parámetros o el tipo de retorno de la función o dejar que el compilador infiera el tipo:

```
// 'Named function'
function multiply(x: number, y: number): number {
  return x * y;
}

// Anonymous function
let add: (baseValue: number, increment: number) => number = function(x: number, y: number): number { return x + y; };

// Esta parte '(baseValue: number, increment: number) => number' es el tipo de la variable 'add', es decir es de tipo 'j
```

Parámetros opcionales

En JavaScript se pueden omitir parámetros en la llamada de la función mientras que en TypeScript no se puede. Si un parámetro no es obligatorio podemos marcarlo como **parámetro opcional** y así obviarlo en la llamada. Para ello usamos el signo de interrogación '?' después del nombre del parámetro:

```
function getFullName(firstName: string, lastName?: string) {
  if (lastName) {
    return `${firstName} ${lastName}`;
  } else {
    return firstName;
  }
}

console.log(getFullName("Thomas", "Huber"));
console.log(getFullName("Thomas"));
console.log(getFullName()); // Error: firstName parameter missing
```

La única regla cuando usamos parámetros opcionales es que los **parámetros obligatorios se definen en primer lugar** y luego se definen los parámetros opcionales.

Valores por defecto

Hay situaciones en que podemos necesitar que un parámetro tenga un valor por defecto si no se informa un valor en la llamada a la función. Si le asignamos un valor por defecto a un parámetro pasa de ser obligatorio a ser opcional debido a que ya no es obligatorio que sea informado:

```
function getFullName(firstName: string = "John", lastName?: string) {
  if (lastName) {
    return `${firstName} ${lastName}`;
  } else {
    return firstName;
  }
}

console.log(getFullName("John", "Doe"));
console.log(getFullName("John"));
console.log(getFullName()); // Ahora si podemos hacer una llamada sin parámetros, ya que el obligatorio tiene valor por defecto
console.log(getFullName(undefined, "Doe"));
```

Los **parámetros con valor por defecto** pueden estar al principio. En ese caso, hacemos la llamada asignando `undefined`.

Si marcamos un parámetro como **opcional y le asignamos un valor por defecto** el compilador arrojará un error en tiempo de compilación:

```
// ¡¡INCORRECTO!!
function getFullName(firstName?: string = "John") {
  // ...
}
```

Número variable de parámetros

En determinadas situaciones podemos necesitar que una función acepte un número variable de parámetros. Al igual que Java o Kotlin, TypeScript permite el paso de un número variable de parámetros. En TypeScript se llama *'rest parameters'* y se indica mediante tres puntos (...) delante del nombre del parámetro:

```
function getFullName(firstName: string, ...moreNames: string[]) {
  return firstName + " " + moreNames.join(" ");
}

// Podemos realizar la llamada a la función pasándole todos los parámetros que sean necesarios
console.log(getFullName("Thomas"));
console.log(getFullName("Thomas", "Huber"));
```

```
console.log(getFullName("Thomas", "Claudius", "Huber"));
console.log(getFullName("Thomas", "Claudius", "Huber", "Developer"));
```

En el ejemplo aunque parece que la función acepta un array de strings, los puntos (...) indican que lo que acepta es un número variable de parámetros de tipo `string`. Dentro de la función este número variable de parámetros se manejará como un array de cadenas.

Si tenemos un array de cadenas, podemos hacer la llamada a la función añadiendo los tres puntos (...) delante del nombre del array:

```
let additionalNames: string[] = ["Claudius", "Huber", "Developer"];
console.log(getFullName("Thomas", ...additionalNames));
```

'Async' y 'Await'

Para hacer el código asíncrono más fácil de escribir, TypeScript tiene soporte para *'async'* y *'await'* al igual que otros lenguajes como C# desde la versión 1.7:

```
function delay(ms: number) {
    return new Promise(resolve => setTimeout(resolve, ms));
}
```

Cuando tenemos un objeto `Promise` podemos usar la palabra clave `await` para esperar al objeto `Promise`. La función donde se utilice `await` tiene que ser marcada como asíncrona con la palabra clave `async`:

```
async function main() {
    await delay(1000);
    console.log("This");

    await delay(1000);
    console.log("is ");

    await delay(1000);
    console.log("ASYNC!");
}
main();
```

Módulos

Los **módulos** sirven para estructurar el código en múltiples ficheros `.ts` en vez de escribir todo el código en un único fichero. Cada fichero tendrá su propio ámbito cuando se usan módulos por lo que hay que exportar explícitamente clases o variables para luego ser importadas y utilizadas en otros ficheros.

Los módulos se incluyen de forma nativa en la especificación ES2015 y por tanto también están disponibles en TypeScript. Con TypeScript se pueden compilar los módulos para ES5 o ES3.

Además de los módulos, TypeScript tiene soporte para los *'namespaces'* o espacios de nombre que tiene un objetivo similar. Dado que los espacios de nombre es un concepto de TypeScript y los módulos forman parte de ES2015 se recomienda el uso de los módulos. En Angular también se utiliza el concepto de módulos.

Los navegadores no tienen la capacidad de cargar los módulos por sí mismos, por lo que se necesita un *'module loader'*. Un cargador de módulos recorre todas las dependencias del módulo raíz. Según las declaraciones de importación, un cargador de módulos encontrará todos los archivos `.js` necesarios y los cargará en consecuencia.

Hay muchos formatos de módulos ('es2015', 'commonjs', 'system', 'amd', 'umd'). Dependiendo del cargador de módulos utilizado, tendremos que indicar el formato en el fichero `tsconfig.json` para que sean compatibles:

```
{
  "compilerOptions": {
    "module": "es2015",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false
  }
}
```

Basics

Un módulo es un fichero `.ts` si tiene al menos un `import` o un `export` en el nivel raíz del fichero.

La ventaja de aplicar el concepto de módulos es que en los módulos el código se ejecuta en su ámbito y no de forma global, lo que significa que variables, funciones, clases, etc... declaradas dentro de un módulo **sólo** son visibles dentro de ese módulo hasta que no se exportan de forma explícita. A la inversa, para utilizar variables, clases, funciones, etc.. que han sido exportadas en otro módulo tienen que ser importadas para poder ser utilizadas.

Un fichero `.ts` que no contiene ningún `export` o `import` a nivel de fichero se considera un *'script'* cuyo contenido es de ámbito global y por tanto está disponible de forma general, incluso también para los módulos.

Export

Cualquier declaración de variable, función, clase, alias o interfaz puede ser exportado añadiendo la palabra clave `export` :

```
// ---- friends.ts
class Friend {
  constructor(public firstName: string) {}
}

export class Developer extends Friend {
  constructor(firstName: string, public lastName: string) {
    super(firstName);
  }
}
```

En el ejemplo tenemos la clase `Developer` que hereda de `Friend`. La clase `Developer` se exporta dado que hemos utilizado la palabra clave `export`. Como tenemos una clase que se exporta, el fichero `friends.ts` es un módulo. La clase `Friend` no se exporta y por tanto sólo es visible dentro de su módulo, o lo que es lo mismo, en el fichero.

Además de clases e interfaces, también se puede exportar variables y/o funciones:

```
// ---- friends.ts
class Friend {
  constructor(public firstName: string) { }
}

export let FRIENDS: Friend[] = [
  new Friend("Sara"),
  new Friend("Anna"),
  new Friend("Thomas")];

export function printFriend(friend: Friend){
  console.log(friend.firstName);
}
```



```
// -----

// ---- main.ts
import { FRIENDS, printFriend } from './friends';

for (let friend of FRIENDS) {
  printFriend(friend);
}

// -----
```

Export aliases

En determinadas situaciones podemos utilizar un *alias* para nombrar la variable, clase, etc... que va a ser exportada con la palabra clave `as` para por ejemplo evitar conflictos de nombres o evitar que sean visibles los nombres reales:

```
// ...

class Developer extends Friend {
  constructor(firstName: string, public lastName: string) {
    super(firstName);
  }
}

export {Developer as Coder};
```

Cuando en la exportación se utiliza un alias, cuando importemos el módulo en otro módulo sólo será visible el alias y no los nombres reales:

```
// ---- friends.ts
export {Developer as Coder};
// -----

// ---- main.ts
import { Coder } from './friends';

let dev = new Coder("John"); // 'Developer' classname is not visible
// -----
```

Export multiple types

Hasta ahora hemos exportado una única entidad pero es posible exportar varias clases dentro del mismo fichero `.ts`. Una forma sería exportar cada clase de forma individual o hacer una única exportación múltiple:

```
class Friend {
  // ...
}

/* export */ class Developer extends Friend {
  // ...
}

/* export */ class Skateboarder extends Friend {
  // ...
}

// Exportación múltiple en un mismo fichero
export { Developer, Skateboarder };
```

Default export

Cada módulo puede exportar opcionalmente una exportación predeterminada o por defecto. Esta exportación predeterminada se indica con la palabra clave `default` y sólo puede haber una exportación por defecto en un módulo:

```
// ---- friends.ts
class Friend {
  constructor(public firstName: string) { }
}

export default class Developer extends Friend {
  // ...
}
```

Para importar un módulo por defecto no se necesitan las llaves ('{}') ni es necesario usar el nombre empleado en la exportación:

```
// ---- main.ts
import Coder from './friends';

let prog = new Coder("John");
```

En un mismo fichero puede haber una exportación por defecto y otras exportaciones:

```
// ---- friends.ts
export class Friend {
  constructor(public firstName: string) { }
}

export default class Developer extends Friend {
  // ...
}
// -----

// ---- main.ts
import Coder, {Friend} from './friends';

let prog = new Coder("John");
// -----
```

Import

Para poder utilizar la clase `Developer` deberemos importarla:

```
// ---- main.ts
import { Developer } from './friends'; // Declaración 'import' con el path del fichero. No es necesario indicar la extensión

let dev = new Developer("John", "Doe");
console.log(dev.firstName); // Prints 'John'
```

Ahora que el fichero `main.ts` tiene una declaración `import`, también se considera un módulo.

Import aliases

Al igual que en la exportación, podemos usar *alias* para realizar la importación para, por ejemplo, evitar conflictos de nombres o mejorar la legibilidad del código:

```
import { Developer as Programmer } from './friends';

var prog = new Programmer("John");
```

Import multiple types

Cuando importamos múltiples tipos que provienen de un mismo módulo, los separamos con comas:

```
import { Developer, Skateboarder } from './friends';
```

En el caso de que sean muchos tipos, podemos optar por realizar la importación del módulo completo usando (*). En ese caso tendremos que utilizar un alias para poder hacer referencia a las clases importadas:

```
import * as Friends from './friends';

var dev = new Friends.Developer("John");

var boarder = new Friends.Skateboarder("Foo");
```

Declaration Files

Cuando se utiliza una biblioteca JavaScript existente, TypeScript no conoce los tipos ya que JavaScript no tiene tipos. Sin tipos, no se obtienen errores en tiempo de compilación ya que TypeScript no puede realizar comprobaciones de tipos.

Es por eso que TypeScript admite [archivos de declaración](#) para bibliotecas JavaScript existentes. El archivo de declaración es un archivo TypeScript normal que por convención termina con `.d.ts` y contiene las declaraciones de tipo para dicha biblioteca.

Por ejemplo, pongamos que tenemos una pequeña biblioteca Javascript con una única función:

```
// ---- myLibrary.js
function printFirstName(friend) {
    document.write("Firstname is " + friend.firstName);
}
```

De forma que podemos utilizar dicha biblioteca y el método que contiene en una página web:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Getting started with TypeScript</title>
    <script src="myLibrary.js"></script>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

En TypeScript podemos utilizar una librería Javascript dado que TypeScript es un superconjunto de Javascript y el código TypeScript se compila en código Javascript:

```
// ---- main.ts
let friend = { firstName: "Thomas" };
printFirstName(friend);
```

Este código funciona pero no hay ningún tipo y además TypeScript muestra un error ya que no puede encontrar la función `printFirstName(friend)` en tiempo de compilación. Aún así, si compilamos el código la función sí que existe en tiempo de ejecución y el código funciona perfectamente.

Para deshacerse del error y obtener una escritura estática que permita pasar un parámetro correcto a la función `printFirstName(friend)`, podemos declarar la función `printFirstName(friend: Friend)` incluyendo una interfaz para su parámetro, como por ejemplo `Friend`.

Declarar una función

Podemos declarar una función en TypeScript usando la palabra clave `declare`. De esta forma TypeScript conoce la función, aunque esté implementada en otro sitio como un biblioteca de terceros. En el ejemplo la función `printFirstName(friend)` se encuentra implementada en el fichero `myLibrary.js` pero se declara en el fichero `main.ts`. Además, con el uso de una interfaz, el compilador puede realizar la comprobación de tipos generando un error en tiempo de compilación:

```
// ---- main.ts
interface Friend {
  firstName: string;
}

// Declaramos la función para que TypeScript conozca su existencia y su firma
declare function printFirstName(friend: Friend): void;

let friend = { firstName: "Thomas" };
printFirstName(friend);
```

Declaraciones en bibliotecas de NPM

Cuando se usan bibliotecas de terceros desde *NPM* en TypeScript tampoco disponemos de los tipos.

Como ejemplo, si queremos usar la biblioteca *'lodash'* en nuestro proyecto, primero la instalamos vía *NPM*:

```
npm install lodash --save
```

A continuación, para utilizar alguna función como por ejemplo la función `range` de la biblioteca *'lodash'*, se importa en el fichero `.ts` que contiene nuestro código:

```
// ---- main.ts
import { range } from 'lodash';

let chapters = range(1, 12);
for (let num in chapters) {
  console.log(num);
}
```

TypeScript no dispone de los tipos con lo cual TypeScript asume que `range(x: any, y: any): any`. Además, tampoco tenemos disponible el autocompletado de IDE ni la documentación, etc...

Para ello podemos instalar la declaración de tipos de la biblioteca *'lodash'* disponible en *NPM*. En *NPM* están las declaraciones de tipos de la mayoría de bibliotecas de terceros. Se puede consultar el listado desde esta [página](#) o buscar en el directorio de *NPM*. Microsoft tiene un [repositorio](#) de definición de tipos.

La declaración de tipos es un fichero que por convención es `d.ts`. Por ejemplo el fichero de *'lodash'* una vez instalado se encuentra en `node_modules/@types/lodash/index.d.ts`. La declaración de tipos se instala vía *NPM*:

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

Una vez instalado TypeScript y el IDE utilizado, como puede ser Visual Studio Code, ahora ya conocen la firma de la función `range()` que es `range(start: number, end: number, step?: number): number[]` lo que permite el autocompletado y la visualización de la documentación.

Declaraciones en bibliotecas propias

Para el caso de bibliotecas propias, el compilador de TypeScript puede generar el fichero de declaración de tipos.

Si por ejemplo tenemos la siguiente biblioteca:

```
// ---- main.ts
interface Friend {
  firstName: string;
}

function printFirstName(friend: Friend) {
  document.write(friend.firstName);
}
```

Para generar el fichero de declaración de tipos, se indica en el fichero `tsconfig.json`:

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

De esta forma, el compilador TypeScript no sólo generará el fichero `main.js` a partir del fichero `main.ts` sino que también generará el fichero `main.d.ts` que contendrá la declaración de la función y su tipo:

```
// ---- main.d.ts
interface Friend {
  firstName: string;
}

declare function printFirstName(friend: Friend): void;
```

Resumen

```

// Existen 3 tipos primitivos en TypeScript
var isDone: boolean = false;
var lines: number = 42; // números enteros y decimales
var name: string = "Anders";

// Cuando es imposible de saber, tenemos el tipo "Any"
var notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okey, definitivamente un boolean

// Para colecciones, hay matrices de tipos y matrices genéricas
var list: number[] = [1, 2, 3];
// Alternativamente, usando La matriz genérica
var list: Array<number> = [1, 2, 3];

// Para enumeradores:
enum Color {
    Red,
    Green,
    Blue
}
var c: Color = Color.Green;

// Finalmente, "void" es usado para el caso especial de una función que no retorna nada
function bigHorribleAlert(): void {
    alert("I'm a little annoying box!");
}

// Las funciones son ciudadanos de primera clase, soportan La sintáxis lambda "fat arrow" y
// usan el tipo inferencia

// Lo siguiente es equivalente, La misma firma será inferida por el
// compilador, y el mismo JavaScript será emitido
var f1 = function(i: number): number {
    return i * i;
};
// Retorna tipo inferido
var f2 = function(i: number) {
    return i * i;
};
var f3 = (i: number): number => {
    return i * i;
};
// Retorna tipo inferido
var f4 = (i: number) => {
    return i * i;
};
// Retorna tipo inferido, one-liner significa que no es necesario que regresen palabras claves
var f5 = (i: number) => i * i;

// Las interfaces son estructurales, todo lo que tenga las propiedades cumple con la interfaz
interface Person {
    name: string;
    // Propiedades opcionales, marcadas con un "?"
    age?: number;
    // Y por supuesto funciones
    move(): void;
}

// Objeto que implementa la interfaz "Persona"
// Puede ser tratada como Persona ya que posee las propiedades name y move
var p: Person = { name: "Bobby", move: () => {} };
// Objetos que tienen propiedades opcionales:
var validPersona: Person = { name: "Bobby", age: 42, move: () => {} };
// No es una persona porque su edad no es un número
var invalidPersona: Person = { name: "Bobby", age: true };

// Las interfaces también pueden describir un tipo de función
interface SearchFunc {
    (source: string, subString: string): boolean;
}

```

```

}
// Solo Los tipos de parámetros son importantes, Los nombres no son importantes.
var mySearch: SearchFunc;
mySearch = function(src: string, sub: string) {
    return src.search(sub) != -1;
};

// Clases - Los miembros son públicos por defecto
class Point {
    // Properties
    x: number;

    // Constructor - Las palabras clave public/private en este contexto generarán
    // un código boiler plate para la propiedad y la inicialización en el constructor.
    // En este ejemplo, "y" debe ser definida al igual que "x" lo es, pero con menos código
    // También son soportados valores por defecto

    constructor(x: number, public y: number = 0) {
        this.x = x;
    }

    // Funciones
    dist() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }

    // Miembros estáticos
    static origin = new Point(0, 0);
}

var p1 = new Point(10, 20);
var p2 = new Point(25); //y será 0

// Herencia
class Point3D extends Point {
    constructor(x: number, y: number, public z: number = 0) {
        super(x, y); // Un llamado explícito al constructor de la super clase es indispensable
    }

    // Sobrescribir
    dist() {
        var d = super.dist();
        return Math.sqrt(d * d + this.z * this.z);
    }
}

// Módulos, Los "." pueden ser usados como separadores para los submódulos
module Geometry {
    export class Square {
        constructor(public sideLength: number = 0) {}
        area() {
            return Math.pow(this.sideLength, 2);
        }
    }
}

var s1 = new Geometry.Square(5);

// Un alias local para referirse a un módulo
import G = Geometry;

var s2 = new G.Square(10);

// Genéricos
// Clases
class Tuple<T1, T2> {
    constructor(public item1: T1, public item2: T2) {}
}

// Interfaces
interface Pair<T> {

```

```
    item1: T;
    item2: T;
}

// Y funciones
var pairToTuple = function<T>(p: Pair<T>) {
    return new Tuple(p.item1, p.item2);
};

var tuple = pairToTuple({ item1: "hello", item2: "world" });

// Incluyendo referencias a un archivo de definición:
/// <reference path="jquery.d.ts" />
```

Enlaces de interés

- <https://www.typescriptlang.org/>
- <https://www.typescriptlang.org/docs/>
- <https://roadmap.sh/typescript>
- <https://learnxinyminutes.com/docs/es-es/typescript-es/>
- <https://goalkicker.com/TypeScriptBook2/>
- <https://github.com/dzharii/awesome-typescript>
- <https://github.com/semlinker/awesome-typescript>
- <https://github.com/type-challenges/type-challenges>
- <https://the-algorithms.com/language/typescript>
- <https://www.tutorialsteacher.com/typescript>
- <https://learntypescript.dev/>
- <https://www.typescripttutorial.net/>
- <https://devhints.io/typescript>
- <https://www.w3schools.com/typescript/index.php>
- <https://basarat.gitbook.io/typescript/>

Licencia



Esta obra está bajo una [licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional](https://creativecommons.org/licenses/by-sa/4.0/).