**UNIVERSIDAD DEL NORTE**

**FACULTAD DE INGENIERIA**

**A blue and white logo

AI-generated content may be incorrect.**

**TRABAJO PRÁCTICO GRUPAL**

**Asignatura:** Lenguajes Visuales II

**Tema:** TypeScript

**Integrantes:**

1. Alfonso Martínez
2. Alcides Monges
3. Giovanni Rojas
4. Jazmín Zorrilla

**Carrera:** Ingeniería Informática

**Curso:** Segundo **Sección:** A **Turno:** Noche

**Docente:** Ing. Milciades González Domínguez

**Fecha de entrega:** 03/11/2025

**Asunción – Paraguay**

**2025**

# Integrantes del grupo

1. Alfonso Martínez Portillo
2. Alcides Fernando Monges Chávez
3. Giovanni Manuel Rojas Giménez
4. Jazmín Del Cielo Betharrán Zorrilla Maciel

# Índice de Contenido

[Introducción a TypeScript 4](#_Toc212924108)

[**¿Qué es TypeScript? 4**](#_Toc212924109)

[**¿Por qué usar TypeScript? 4**](#_Toc212924110)

[ Tipado estático e interfaz de tipo 4](#_Toc212924111)

[ Detección temprana de errores 5](#_Toc212924112)

[❖ Capacidad de mantenimiento y escalabilidad del código 5](#_Toc212924113)

[**Diferencias clave entre TypeScript y JavaScript 5**](#_Toc212924114)

[1. Tipado estático vs tipado dinámico 5](#_Toc212924115)

[2. Errores en tiempo de compilación y errores en tiempo de ejecución 6](#_Toc212924116)

[3. Inferencia de tipos y anotaciones de tipos 6](#_Toc212924117)

[4. Programación orientada a objetos 6](#_Toc212924118)

[5. Herramientas y soporte IDE 7](#_Toc212924119)

[6. Compatibilidad con versiones anteriores e interoperabilidad 7](#_Toc212924120)

[**Instalación de TypeScript y configuración básica 8**](#_Toc212924121)

[¿Cómo instalar TypeScript en el sistema? 8](#_Toc212924122)

[¿Cómo mejorar la experiencia de desarrollo con extensiones? 9](#_Toc212924123)

[**Compilador de TypeScript (tsc) 9**](#_Toc212924124)

[¿Cómo inicializar un proyecto de TypeScript? 9](#_Toc212924125)

[Compilador versus servicio de lenguaje 10](#_Toc212924126)

[Tipos Básicos 11](#_Toc212924127)

[**Tipos primitivos 11**](#_Toc212924128)

[**Tipos avanzados de TypeScript 12**](#_Toc212924129)

[Tipo any 13](#_Toc212924130)

[Tipo unknown 13](#_Toc212924131)

[Tipo never 13](#_Toc212924132)

[Tipo void 14](#_Toc212924133)

[**Arrays 14**](#_Toc212924134)

[Tuplas (tuples) 14](#_Toc212924135)

[Uniones de tipos 15](#_Toc212924136)

[Funciones en Typescript 15](#_Toc212924137)

[**Funciones Tipadas 15**](#_Toc212924138)

[**Parámetros Opcionales 16**](#_Toc212924139)

[**Funciones Flecha: 16**](#_Toc212924140)

[Aplicaciones de las funciones flecha 19](#_Toc212924141)

[¿Qué es una función de devolución de llamada (callback)? 19](#_Toc212924142)

[**Uso de Función Void 21**](#_Toc212924143)

[¿Cómo usar void con tipos de función? 21](#_Toc212924144)

[Conclusión 23](#_Toc212924145)

[Bibliografía 24](#_Toc212924146)

# Introducción a TypeScript

## ¿Qué es TypeScript?

TypeScript es un lenguaje de programación de código abierto desarrollado por Microsoft, como un superconjunto, que extiende las capacidades de JavaScript. Está pensado para desarrollar proyectos grandes y complejos de manera más ordenada.

Los proyectos comunes escritos en JavaScript también son programas TypeScript válidos, y el lenguaje se puede usar para crear aplicaciones JavaScript para la ejecución del lado del servidor y no solo para la ejecución del lado del cliente.

TypeScript admite archivos que pueden incluir datos de tipo de bibliotecas de JavaScript actuales, al igual que los archivos de encabezado de C++ pueden definir la construcción de archivos de objetos actuales, lo que permite que otros programas ejerzan los valores definidos en los archivos como si fueran entidades de TypeScript tipificadas estáticamente.

Hay archivos de encabezado de terceros para bibliotecas predominantes como D3.js, jQuery y MongoDB. También se puede acceder a los encabezados de TypeScript para los módulos básicos de Node.js, lo que permite el desarrollo de programas de Node.js en TypeScript.

Durante el proceso de compilación, el código TypeScript no se convierte en código máquina, sino que se transcompila a JavaScript, permitiendo su ejecución en cualquier entorno compatible con este lenguaje. El compilador de TypeScript está registrado bajo la licencia Apache 2.0.

Esencialmente, TypeScript es JavaScript con la capacidad adicional de usar tipos estáticos cuando sea necesario.

## ¿Por qué usar TypeScript?

TypeScript se basa en JavaScript agregando escritura estática, lo que ayuda a los desarrolladores a escribir código más seguro y predecible.

No solo mejora la calidad del código al detectar errores desde el principio, sino que también mejora la experiencia de desarrollo con mejores herramientas, intenciones más claras y una refactorización más fácil. A continuación se presentan algunos de los beneficios clave.

### Tipado estático e interfaz de tipo

La tipificación estática de TypeScript permite a los desarrolladores detectar errores desde el principio al inferir o definir explícitamente los tipos. Por ejemplo, en JavaScript, si lo declaras y luego lo reasignas a true, JavaScript no lo compilará, lo que podría generar un comportamiento confuso o no deseado.

let iceCream = “Chocolate”

TypeScript nos da dos opciones para tratar este problema. La primera es simplemente dejar que TypeScript infiera el tipo en función del uso. En el ejemplo, TypeScript podrá inferir que la variable iceCream debe ser una cadena y no permitirá asignarle un tipo diferente.

La segunda forma es decirle a TypeScript de qué tipo debe ser la variable. Asignamos explícitamente un tipo.

### Detección temprana de errores

Una de las mayores fortalezas de TypeScript es su capacidad para detectar errores en el momento en que se compila el código, pero antes de que se convierta. Este sistema puede identificar errores lógicos, operaciones no válidas y uso inadecuado de variables y funciones que podrían no encontrarse con la detección de errores estándar.

### Capacidad de mantenimiento y escalabilidad del código

TypeScript alienta a los desarrolladores a escribir código predecible y autodocumentado, lo que a su vez mejora la legibilidad de otros desarrolladores que trabajan en el mismo código y facilita la colaboración entre equipos grandes.

Además, la refactorización se vuelve más segura y eficiente, ya que TypeScript puede realizar un seguimiento de los cambios en los tipos en todo el proyecto. Por ejemplo, si cambia el nombre de una propiedad o cambia la firma de una función, TypeScript le avisará de todos los lugares que dependen de ella para evitar errores de lógica rota o de tiempo de ejecución.

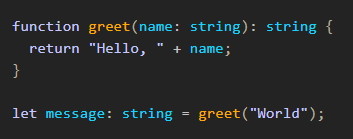
## Diferencias clave entre TypeScript y JavaScript

### Tipado estático vs tipado dinámico

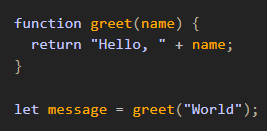
Una de las diferencias más significativas entre TypeScript y JavaScript son sus sistemas de tipos:

* **TypeScript:** Introduce tipado estático, lo que significa que puedes definir los tipos de variables, parámetros de funciones y valores de retorno. Esto permite la detección temprana de errores durante el desarrollo.
* **JavaScript:** Tiene tipos dinámicos, lo que significa que los tipos se determinan en tiempo de ejecución. Esto puede generar errores en tiempo de ejecución si las variables no se utilizan como se espera.

Ejemplo de TypeScript



Ejemplo de JavaScript



### Errores en tiempo de compilación y errores en tiempo de ejecución

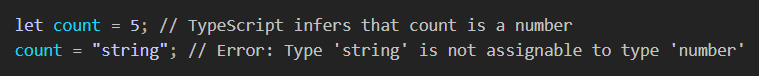
La tipificación estática de TypeScript permite detectar errores en el momento de la compilación, antes de que se ejecute el código. Esto puede ayudar a identificar posibles problemas en las primeras fases del proceso de desarrollo:

* **TypeScript:** Se detectan errores relacionados con desajustes de tipos y otros problemas durante la compilación, lo que ayuda a prevenir errores comunes.
* **JavaScript:** Los errores generalmente se descubren en tiempo de ejecución, lo que puede hacer que la depuración sea más desafiante ya que los problemas pueden aparecer solo cuando se ejecuta el código.

### Inferencia de tipos y anotaciones de tipos

TypeScript ofrece funciones de tipo avanzadas, como inferencia de tipo y anotaciones de tipo:

* **TypeScript:** Puede especificar tipos explícitamente mediante anotaciones de tipos o dejar que TypeScript los infiera en función del código. Esto mejora la claridad del código y reduce los errores.
* **JavaScript:** No tiene anotaciones de tipos integradas ni inferencias. Los desarrolladores dependen de convenciones y verificaciones en tiempo de ejecución para administrar los tipos.

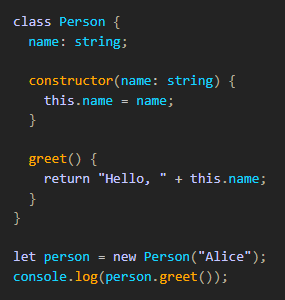


### Programación orientada a objetos

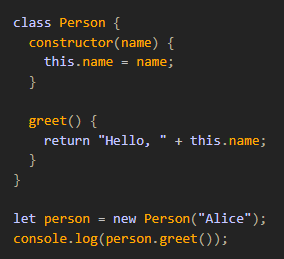
TypeScript proporciona un mejor soporte para los principios de programación orientada a objetos (POO):

* **TypeScript:** Incluye características como clases, interfaces y herencia, que facilitan el trabajo con conceptos de programación orientada a objetos.
* **JavaScript:** Admite la programación orientada a objetos, pero lo hace con herencia basada en prototipos y una sintaxis menos formal. Las características de programación orientada a objetos de TypeScript están más alineadas con los lenguajes tradicionales basados ​​en clases.

Ejemplo de clase TypeScript



Ejemplo de clase de JavaScript



### Herramientas y soporte IDE

TypeScript generalmente ofrece un mejor soporte de herramientas en comparación con JavaScript:

* **TypeScript:** Muchos IDE y editores proporcionan funciones mejoradas como autocompletado, verificación de tipos y documentación en línea para proyectos TypeScript.
* **JavaScript:** Si bien los IDE modernos también admiten JavaScript, las funciones de verificación de tipos y de autocompletado no son tan sólidas como las disponibles para TypeScript.

### Compatibilidad con versiones anteriores e interoperabilidad

TypeScript está diseñado para ser compatible con el código JavaScript existente:

* **TypeScript:** El código TypeScript se puede compilar en JavaScript, lo que hace posible el uso de TypeScript en proyectos con bases de código JavaScript existentes.
* **JavaScript:** El código JavaScript se puede integrar en proyectos TypeScript ya que TypeScript permite una adopción gradual.

Observando las diferencias entre TypeScript y JavaScript, podemos decir que ambas tecnologías, si bien son cercanas, tienen sus ventajas y desventajas.

JavaScript es más adecuado cuando se tiene un pequeño equipo trabajando en pequeños proyectos web. Por otro lado, si se cuenta con un equipo más experimentado, con buenos conocimientos o si se trata de la gestión de un proyecto más complejo, TypeScript es una mejor opción.

Para proyectos más pequeños, el uso de TypeScript puede ser una sobrecarga, ya que traducir el código a JavaScript lleva tiempo, lo que es un paso adicional para la representación.

Dado que JavaScript se ejecuta directamente en el navegador, es más fácil actualizar o depurar el código para scripts pequeños, mientras que en el caso de TypeScript, se necesitará un IDE [y](https://www-codeur-com.translate.goog/blog/meilleurs-environnement-de-developpement-javascript/?_x_tr_sl=auto&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=wapp) una configuración adecuada para ejecutar el código.

## Instalación de TypeScript y configuración básica

TypeScript se puede instalar a través de tres rutas de instalación dependiendo de cómo se desee usar:

* un módulo npm,
* un paquete NuGet
* una extensión de Visual Studio.

Si se utiliza Node.js, es necesaria la versión npm. Si MSBuild es el utilizado para el proyecto, entonces se debe optar por el paquete NuGet o la extensión de Visual Studio.

### ¿Cómo instalar TypeScript en el sistema?

Antes que nada, es fundamental verificar que se cuenta con los requisitos previos necesarios. TypeScript depende completamente de Node.js y NPM para su funcionamiento, por lo que estos deben estar instalados en el sistema antes de proceder.

Para comprobar si ya se tienen estas herramientas, hay que ejecutar los siguientes comandos en la terminal:

node -v

npm -v

Estos comandos mostrarán las versiones instaladas de Node.js y NPM respectivamente. Si se obtiene una respuesta con números de versión, significa que ya están instalados y es posible continuar. En caso contrario, deben ser instalados antes de proseguir.

Una vez confirmado, se procede a instalar TypeScript globalmente en el sistema mediante el siguiente comando:

npm install -g typescript

Es posible que los permisos de administrador sean necesarios para realizar esta instalación global. En sistemas basados en Unix (como Linux o macOS) se utiliza:

sudo npm install -g typescript

Para verificar que la instalación se ha completado correctamente, se ejecuta:

tsc -v

Este comando debería mostrar la versión de TypeScript instalada en el sistema, confirmando que todo está listo para comenzar a trabajar.

### ¿Cómo mejorar la experiencia de desarrollo con extensiones?

Visual Studio Code es el editor recomendado para trabajar con TypeScript, ya que ofrece un excelente soporte para este lenguaje. Para optimizar aún más la experiencia de desarrollo, es recomendable instalar la extensión "JavaScript and TypeScript Nightly" desarrollada por Microsoft. Esta extensión proporciona:

* Resaltado de sintaxis mejorado
* Autocompletado inteligente
* Detección de errores en tiempo real
* Sugerencias de código más precisas

Para instalarla, simplemente se debe:

1. Abrir la sección de extensiones en VS Code (icono en la barra lateral)
2. Buscar "TypeScript"
3. Localizar "JavaScript and TypeScript Nightly" de Microsoft
4. Hacer clic en instalar

Es importante verificar que el autor sea Microsoft para garantizar que se está instalando la extensión oficial y segura.

## Compilador de TypeScript (tsc)

### ¿Cómo inicializar un proyecto de TypeScript?

La forma más sencilla de instalar TypeScript es a través de npm, el gestor de paquetes de Node.js. Con TypeScript instalado y el editor configurado, es momento de inicializar el proyecto. Para ello, es necesario navegar hasta la carpeta en la que se desea crear el proyecto y ejecutar:

tsc --init

Este comando creará un archivo **tsconfig.json** en el directorio actual. Este archivo es fundamental para la configuración de TypeScript en el proyecto, ya que define cómo se comportará el compilador.

El archivo generado contiene numerosas opciones comentadas que son personalizables. Las configuraciones básicas, como el target (versión de JavaScript a la que se compilará) y los módulos, aparecen sin comentar por defecto.

{

"compilerOptions":

{

"target": "es2016",

"module": "commonjs",

// Muchas más opciones comentadas...

}

}

Es recomendable explorar estas opciones para entender qué opciones son personalizables en el proyecto. Algunas configuraciones importantes incluyen:

* **target**: Define la versión de JavaScript a la que se compilará el código
* **module**: Especifica el sistema de módulos a utilizar
* **strict**: Activa un conjunto de comprobaciones de tipo estrictas
* **outDir**: Directorio donde se generarán los archivos JavaScript compilados

**Compilando TypeScript**

Visual Studio Code es compatible con el lenguaje, pero no incluye el compilador TypeScript. Por lo que se necesitará instalar el compilador, ya sea globalmente o en el espacio de trabajo, para transpilar el código fuente TypeScript a JavaScript ( tsc HelloWorld.ts).

La forma más sencilla de instalar TypeScript es a través de npm, el gestor de paquetes de Node.js. Si se tiene npm instalado, se puede instalar TypeScript globalmente ( -g ) mediante:

npm install -g typescript

Se puede probar la instalación consultando la versión o la ayuda.

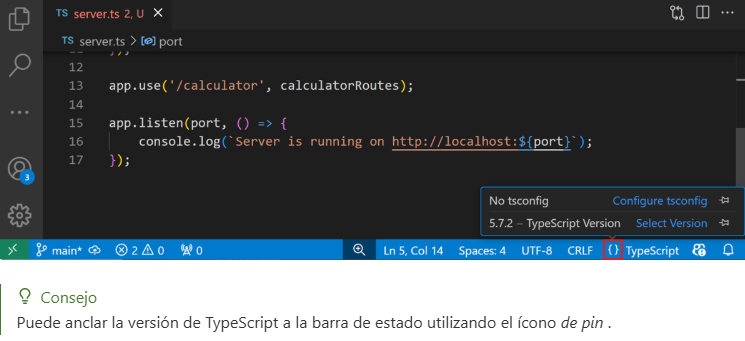
tsc --version

tsc --help

Otra opción es instalar el compilador TypeScript localmente en el proyecto **(npm install --save-dev typescript)** así se tiene el beneficio de evitar posibles interacciones con otros proyectos TypeScript.

### Compilador versus servicio de lenguaje

Es importante tener en cuenta que el servicio de lenguaje TypeScript de VS Code es independiente del compilador TypeScript instalado. Para ver la versión de TypeScript de VS Code se puede observar en la barra de estado del lenguaje.



**tsconfig.json:**

Normalmente, el primer paso en cualquier proyecto nuevo de TypeScript es agregar un **tsconfig.jsonarchivo**. Un **tsconfig.jsonarchivo** define la configuración del proyecto de TypeScript, como las opciones del compilador y los archivos que deben incluirse. Para ello, hay que abrir la carpeta donde desea almacenar el código fuente y agregar un nuevo archivo llamado **tsconfig.json.** Una vez en este archivo, IntelliSense **( Ctrl+Espacio )** ayudará.



Un **tsconfig.jsonejemplo** simple para ES5, módulos **CommonJS** y mapas fuente es el siguiente:

{

"compilerOptions":

{

"target": "ES5",

"module": "CommonJS",

"sourceMap": true

}

}

11

Ahora, al crear un **.tsarchivo** como parte del proyecto, se ofrecerá experiencias de edición enriquecidas y validación de sintaxis.

# Tipos Básicos

## Tipos primitivos

Estos tipos se heredan directamente de JavaScript y son esenciales para definir la estructura básica de datos y garantizar la seguridad de tipos en aplicaciones TypeScript. Si bien *null* y *undefined* pueden asignarse a variables de otros tipos (especialmente si las comprobaciones de nulos estrictas están deshabilitadas o permitidas explícitamente con un tipo de unión), también existen como tipos distintos.

1. **String**: Representa datos textuales. Otro aspecto fundamental de la creación de programas en JavaScript, tanto para páginas web como para servidores, es trabajar con datos textuales. Al igual que en otros lenguajes, usamos el tipo string para referirnos a estos tipos de datos textuales. Al igual que JavaScript, TypeScript también usa comillas dobles o simples para enmarcar datos de cadena.

let greeting: string = "Hello, TypeScript!";

1. **Number:** Representa números enteros y de punto flotante. Al igual que en JavaScript, todos los números en TypeScript son valores de punto flotante o *BigIntegers*. Estos números de punto flotante reciben el tipo *number*, mientras que los *BigIntegers* reciben el tipo *bigint*. Además de los literales hexadecimales y decimales, TypeScript también admite los literales binarios y octales introducidos en ECMAScript 2015.

let age: number = 30;  
let price: number = 99.99;

1. **Boolean:** Representa una entidad lógica y puede tener dos valores *true* o*false*. El tipo de dato más básico es el simple valor verdadero/falso, al que JavaScript y TypeScript llaman *booleanvalor*.

let isActive: boolean = true;

1. **Null**: Representa la ausencia intencional de cualquier valor de objeto. Es un valor primitivo. El tipo *Null* tiene exactamente un valor:*null*.

let user: string | null = null;

1. **Undefined**: Representa una variable declarada, pero sin valor asignado. También es un valor primitivo. Una variable a la que no se le ha asignado un valor tiene el valor *undefined*. Consulta undefined y Undefined para obtener más detalles.

let quantity: number | undefined;

## Tipos avanzados de TypeScript

En TypeScript, los tipos avanzados son formas más complejas y potentes de describir los datos que maneja un programa.

Mientras que los tipos primitivos (como string, number, boolean) sirven para representar valores simples, los tipos avanzados permiten mayor precisión, flexibilidad y control sobre el comportamiento de las variables y funciones.

Los tipos avanzados son herramientas del sistema de tipos de TypeScript que nos ayudan a:

* Definir con mayor exactitud qué tipo de datos puede tener una variable.
* Evitar errores al detectar incompatibilidades de tipo antes de ejecutar el programa.
* Combinar o restringir tipos, según las necesidades del código.

### Tipo any

El tipo *any* es el más flexible en TypeScript, ya que desactiva el sistema de tipos.

Permite asignar valores de cualquier tipo, lo que puede ser útil en casos donde no se conoce el tipo exacto de un dato, pero también puede introducir errores si se abusa de él.

* **Ventaja:** permite flexibilidad en variables con tipos desconocidos.
* **Desventaja:** elimina la seguridad de tipos, similar a usar JavaScript puro.

### Tipo unknown

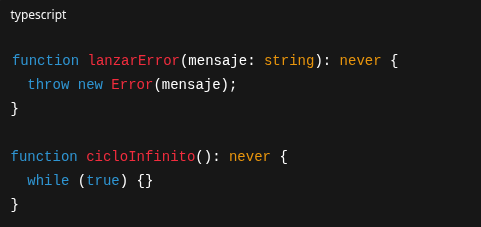
El tipo *unknown* también representa un valor de tipo desconocido, pero a diferencia de *any*, mantiene la seguridad de tipos.

No se puede usar un valor *unknown* directamente sin antes hacer una comprobación de tipo.

* **Ventaja:** más seguro que *any*, ya que obliga al programador a verificar el tipo antes de operar.
* **Desventaja:** requiere comprobaciones adicionales.

### Tipo never

El tipo never representa valores que nunca ocurren. Se usa en funciones que nunca retornan, ya sea porque lanzan un error o porque entran en un bucle infinito.



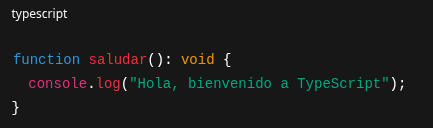
Además, también se usa para indicar que una función no termina correctamente o no devuelve nada.

Es importante resaltar que no se puede asignar ningún valor al tipo *never*.

### Tipo void

El tipo void se utiliza principalmente en funciones que no retornan ningún valor.

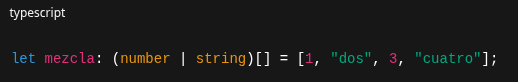
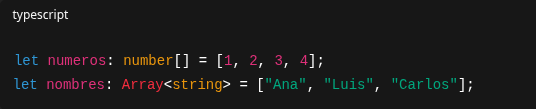
Es equivalente a decir “no devuelve nada”, similar a *void* en C, C++ o Java.



Es utilizado principalmente en funciones que ejecutan acciones pero no devuelven datos.

Una variable de tipo void solo puede contener *undefined* o no tener valor.

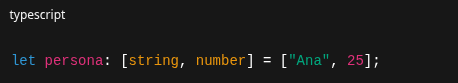
## Arrays

En TypeScript, los arrays permiten almacenar múltiples valores del mismo tipo. Se pueden declarar de dos formas equivalentes:

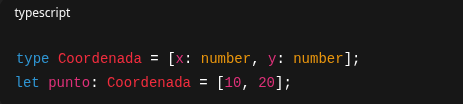
También se pueden crear arrays mixtos usando uniones de tipos:

Su principal ventaja es que proporciona tipado fuerte dentro del arreglo, previniendo errores al acceder o modificar los elementos.

### Tuplas (tuples)

Las tuplas son un tipo especial de array donde se define la cantidad y el tipo exacto de los elementos. A diferencia de los arrays, en una tupla el orden y tipo de cada posición importa.

En este ejemplo, la primera posición debe ser una cadena (string) y la segunda un número (number). También se pueden usar tuplas opcionales o con nombres:



Permite definir estructuras fijas, ideales para datos de posición, pares clave-valor o resultados combinados. Es importante tener en cuenta que, si se modifica la tupla incorrectamente, puede romper el orden de tipos.

### Uniones de tipos

Una unión de tipos permite que una variable pueda ser de más de un tipo. Se define usando el operador | (OR).

También puede usarse en funciones:

Esto nos brinda flexibilidad sin perder el control de tipos. Es por eso que se recomienda combinar con verificaciones de tipo (typeof, instanceof).

# Funciones en Typescript

Las funciones son el componente fundamental de cualquier aplicación en JavaScript. Permiten construir capas de abstracción, imitando clases, ocultando información y módulos. En TypeScript, si bien existen clases, espacios de nombres y módulos, las funciones siguen desempeñando un papel fundamental al describir cómo hacer las cosas.

Para empezar, al igual que en JavaScript, las funciones de TypeScript se pueden crear tanto con nombre como anónimas. Esto permite elegir el enfoque más adecuado para la aplicación, ya sea creando una lista de funciones en una API o una función única para transferir a otra función.

## Funciones Tipadas

Las funciones tipadas en Typescript son funciones en las que se especifican los tipos de los parámetros y del valor que devuelven.

Esto permite que el compilar verifique que la función se use correctamente y evita errores comunes en tiempo de ejecución.

A blue and orange text

AI-generated content may be incorrect.

* ***a: number*** y ***b: number*** -> definen el tipo de parámetros
* ***: number*** después de los paréntesis -> indica el tipo de retorno
* Si uno intenta pasar algo que no sea número, TypeScript mostrará un error antes de ejecutar el código.

El tipado de las funciones sirve para controlar qué tipo de datos entran y salen de la función, haciendo el código más seguro, legible y fácil de mantener, el tipado actúa como una barrera de seguridad que evita pasar datos incorrectos y ayuda a detectar errores antes de ejecutar un código.

## Parámetros Opcionales

En JavaScript, se puede llamar a una función sin pasar argumentos, aunque esta especifique parámetros. Por lo tanto, JavaScript admite los parámetros opcionales por defecto.

En TypeScript, el compilador verifica cada llamada de función y emite un error en los siguientes casos:

* El número de argumentos es diferente del número de parámetros especificados en la función.
* O los tipos de argumentos no son compatibles con los tipos de parámetros de la función.

Debido a que el compilador verifica exhaustivamente los argumentos que pasan, es necesario anotar los parámetros opcionales para indicarle al compilador que no emita un error cuando omite los argumentos.

En TypeScript, los parámetros opcionales son aquellos que no es obligatorio pasarle a una función cuando la llamas. Se definen agregando un signo de interrogación “­?” después del nombre del parámetro.

**Ejemplo básico:**

A screen shot of a computer code

AI-generated content may be incorrect.

* ***saludo?: string*** -> el parámetro saludo es opcional
* Si no se pasa, su valor sería *undefined*
* Dentro de la función se puede verificar si se proporcionó o no.

Los parámetros opcionales en TypeScript permiten que una función sea más flexible, ya que pueden recibir uno o varios argumentos menos sin romper el código.

## Funciones Flecha:

Las funciones flecha son una forma concisa de escribir funciones anónimas en TypeScript. Ofrecen una sintaxis más breve y legible que las declaraciones de funciones tradicionales. También se denominan funciones lambda. Lambda se refiere a las funciones anónimas en programación.

**Sintaxis**

Puedes definir una función de flecha usando una flecha gruesa (=>). Esta flecha se usa para separar los parámetros de la función del cuerpo (declaraciones).

Una función de flecha consta de 3 partes:

* **Parámetros** − Una función puede tener opcionalmente parámetros, *param1, param2, ..., paramN.*
* **La notación de flecha gruesa/notación lambda (=>)** − También se la denomina operador de destino.
* **Declaraciones**: representan el conjunto de instrucciones de las funciones

**Ejemplo, función flecha con una sola declaración:**

A computer screen shot of a number

AI-generated content may be incorrect.

* La función tiene un solo propósito: sumar a y b
* Una sola declaración = la función realiza una acción simple como (a + b)
* Funciones flecha permiten escribir esto de forma compacta y clara.

**Explicación:**

***const sumar = ...***

* Se está declarando una variable llamada sumar.

***(a: number, b: number)***

* Son los parámetros de la función, *a:* *number* → indica que el primer parámetro debe ser un número, *b: number* → el segundo también debe ser un número.
* Esto es el tipado de TypeScript, lo que ayuda a evitar errores.

***: number*** después de los paréntesis

* Indica el tipo de valor que devuelve la función.

***=> { return a + b; }***

* ***=>*** indica que esto es una **función flecha**.
* El cuerpo de la función suma ***a + b*** y lo devuelve con ***return***.

Cuando usarla:

Cuando la función realiza una única operación simple.

**Ejemplo, funciones flecha con múltiples declaraciones**

Se puede escribir varias sentencias entre llaves. También necesitamos escribir la sentencia de retorno para devolver un valor.

A screen shot of a computer code

AI-generated content may be incorrect.

**Explicación:**

1. **(a: number, b: number): number => { ... }**

* Declaramos una función flecha con tipado de parámetros y retorno.

1. **Múltiples declaraciones dentro de { }**

* ***console.log(...)*** para mostrar cada número.
* ***const suma = a + b;*** para realizar la operación.

1. **return suma;**

* Como hay varias declaraciones, necesitamos **return** explícito para devolver un valor.

1. **Uso de la función**

* const resultado = procesarNumeros(5, 3); llama a la función.
* console.log(resultado); muestra el valor retornado.

**Cuando usarlo:**

Cuando la función necesita ejecutar mas líneas de código y una lógica mas compleja, como validaciones o condiciones.

**Ejemplo: Función flecha sin parámetro**

Los parámetros de las funciones flecha son opcionales.

A computer code with colorful text

AI-generated content may be incorrect.

**Explicación:**

1. ***()*** -> Indica que la función no recibe ningún parámetro
2. **: void** -> indica que no devuelve ningún valor
3. **{ console.log(...) }** -> cuerpo de la función.

**Cuando usarlo:**

Cuando no se necesita que la función reciba datos y la función simplemente ejecuta la acción o devuelve un valor fijo.

**Ejemplo: Función flecha con un solo parámetro**

****

**Cuando es conveniente usarlo**

Cuando la función recibe un solo valor, por ejemplo para transformarlo o mostrarlo.

**Ejemplo: Función de flecha con múltiples parámetros**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**Cuando es conveniente usarlo:**

Cuando la función debe de recibir uno o más valores y realiza una operación con esos valores.

### Aplicaciones de las funciones flecha

**Ejemplo:** La flecha funciona como función de devolución de llamada en otra función.

Las funciones flecha son muy utilizadas como funciones de devolución de llamada (también llamadas *callbacks*) porque su sintaxis es más corta y no cambia el contexto de *this*.

### ¿Qué es una función de devolución de llamada (callback)?

Una callback es una función que se pasa como argumento a otra función para que se ejecutedentro de ella.

A screenshot of a computer program

AI-generated content may be incorrect.

* 1. **map()** -> es una función que recorre cada elemento del arreglo números.
  2. **(n: number): number** => *n \* n* → es la función flecha callback, que recibe cada número n y devuelve su cuadrado.
  3. **map** aplica esa función a cada elemento y devuelve un nuevo arreglo con los resultados.

**A screen shot of a computer program

AI-generated content may be incorrect.Ejemplo: Funciones flecha en la clase.**

Las funciones de flecha pueden emplearse como métodos de clase o propiedades de objeto en TypeScript.

**¿Por qué usar funciones flecha en clases?**

En JavaScript/TypeScript, el valor de *this* puede cambiar dependiendo de cómo se invoque el método.

Con una función flecha, el *this* siempre se refiere al objeto que la contiene, incluso si se pasa como callback o se usa fuera del contexto de la clase.

|  |  |  |
| --- | --- | --- |
| **Línea** | **Qué hace** | **Resultado** |
| class Boton { ... } | Define la clase | — |
| constructor(texto) | Inicializa la propiedad texto | "Enviar" |
| click = () => {...} | Define un método flecha que mantiene el this | Evita errores de contexto |
| new Boton("Enviar") | Crea un nuevo botón | miBoton |
| setTimeout(miBoton.click, 1000) | Ejecuta el método después de 1 segundo | “Botón 'Enviar' fue clickeado.” |

**Ejemplo: Implementación de funciones de orden superior**

Las funciones de orden superior, que son funciones que aceptan una o más funciones como argumentos y/o devuelven una función como resultado.

**¿Qué es una función de orden superior?**

Una función de orden superior es aquella que: recibe una función como parámetro, o devuelve una función como resultado.

Este concepto es muy común en JavaScript/TypeScript, especialmente cuando trabajamos con callbacks, promesas o funciones como map, filter, reduce, etc.

A screenshot of a computer program

AI-generated content may be incorrect.

## Uso de Función Void

El tipo void en TypeScript representa la ausencia de un valor. Se utiliza principalmente, en el contexto de funciones, para indicar que una función no retorna ningún valor significativo.

Cuando una función está diseñada para realizar una acción (a veces llamado efecto secundario), como modificar un estado o imprimir un mensaje en la consola. Entonces no necesita devolver un resultado, se especifica el tipo *void*.

### ¿Cómo usar void con tipos de función?

Según la documentación oficial, "*nulo*" se puede expresar de la siguiente manera:

* La ausencia de tener cualquier tipo en absoluto.

En concreto, si se asigna *void* un tipo de función como tipo de retorno, el verificador de tipos ignorará el tipo devuelto por la función.

**Ejemplo:**

****

**Explicación:**

**type *OperacionVoid* = (a: number, b: number) => void;**

* Define un tipo de función que recibe dos números y no devuelve ningún valor (solo ejecuta)
* **mostrarSuma** implementa esa estructura
* El verificador de tipos de TypeScript ignora cualquier valor devuelto, porque el tipo declarado es *void*.

***Ejemplo en la vida real:***

1. Se solicita a una función que imprima un ticket
2. No es necesario que devuelva un valor, solo que haga la acción
3. Esa función sería de tipo *void*

# Conclusión

El aprendizaje de TypeScript permite comprender cómo un sistema de tipado estático puede mejorar significativamente la calidad del código en comparación con JavaScript. A través de su uso, el desarrollador adquiere la capacidad de detectar errores en tiempo de compilación, escribir código más claro y mantener una estructura más ordenada y predecible en los proyectos.

Además, el dominio de los tipos de datos y del tipado en funciones fomenta una programación más segura y eficiente, reduciendo la posibilidad de fallos y facilitando el trabajo en equipo en proyectos de gran escala. En definitiva, TypeScript se consolida como una herramienta esencial para todo desarrollador que busque combinar la flexibilidad de JavaScript con la solidez y escalabilidad de un lenguaje fuertemente tipado.

# Bibliografía

[https://www.webopedia.com/definitions/typescript-definition-meaning/](https://www.webopedia.com/definitions/typescript-definition-meaning/?utm_source=chatgpt.com)

<https://builtin.com/software-engineering-perspectives/typescript>

[https://es.sharpcoderblog.com/blog/typescript-vs-javascript-key-differences-explained](https://es.sharpcoderblog.com/blog/typescript-vs-javascript-key-differences-explained?utm_source=chatgpt.com)

[https://www.kreactiva.com/programacion-web/typescript-vs-javascript-cuales-son-las-diferencias/](https://www.kreactiva.com/programacion-web/typescript-vs-javascript-cuales-son-las-diferencias/?utm_source=chatgpt.com)

<https://platzi.com/cursos/typescript/instalacion-de-nodejs-y-typescript-cli-configuraci/>

<https://code.visualstudio.com/docs/typescript/typescript-compiling>

<https://www.typescriptlang.org/docs>

<https://developer.mozilla.org/>

<https://www.typescriptlang.org/docs/handbook/4>

<https://www.tutorialspoint.com/typescript/typescript_arrow_functions>

<https://www.typescriptlang.org/docs/handbook/functions>

<https://www.luisllamas.es/tipo-void-typescript/>

<https://dev.to/tomoy/what-is-void-in-typescript-2jg7>