



TypeScript : Tipos básicos

Dominar TypeScript es clave para cualquier desarrollador JavaScript moderno. Esta presentación te guiará a través de sus fundamentos, tipos básicos y cómo empezar a usarlo en tus proyectos.

¿Qué es TypeScript y Por Qué Debería Usarlo?

TypeScript es un superconjunto de JavaScript. Eso quiere decir que todo el código JavaScript es válido en TypeScript, pero TypeScript añade algo muy valioso: los tipos de datos

- **Mejora la calidad del código:** Captura errores comunes de programación.
- **Mayor productividad:** Proporciona autocompletado y refactorización avanzada.
- **Código más legible:** Facilita la comprensión de la estructura de los datos.
- **Escalabilidad:** Ideal para proyectos grandes y equipos.



Tipos Básicos en TypeScript: Primitivos

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Number

¿Qué es? Representa números (enteros, decimales, hexadecimales, etc.)

¿Cuándo usarlo? Para edades, precios, coordenadas, contadores

typescript

```
let edad: number = 25;  
let precio: number = 19.99;
```

Tipos Básicos en TypeScript: Primitivos

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

String

¿Qué es? Representa texto, cadenas de caracteres (Admite comillas simples, dobles o backticks.)

¿Cuándo usarlo? Para nombres, mensajes, IDs de texto, URLs, etc.

```
typescript
let nombre: string = "Ana";
let mensaje: string = `Hola ${nombre}`;
```

Tipos Básicos en TypeScript: Primitivos

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Boolean

¿Qué es? Representa valores de verdad: solo puede ser true o false

¿Cuándo usarlo? Para flags, estados de activación/desactivación, condiciones

typescript

```
let esActivo: boolean = true;  
let tienePermiso: boolean = false;
```

Tipos especiales en TypeScript

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Undifined

¿Qué es? Indica que una variable fue declarada pero no se le asignó valor

¿Cuándo usarlo? Para propiedades opcionales o variables no inicializadas

typescript

```
let configuracion: string | undefined;
```

Tipos especiales en TypeScript

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Null

¿Qué es? Representa la ausencia intencional de valor

¿Cuándo usarlo? Cuando quieras indicar explícitamente que "no hay nada"

typescript

```
let usuarioSinSesion: string | null = null;
```

Tipos especiales en TypeScript

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Any

¿Qué es? Desactiva el sistema de tipos - puede ser cualquier cosa

¿Cuándo usarlo? Solo cuando migras código JavaScript o trabajas con librerías sin tipos (¡hay que evitárselo!)

typescript

```
let cualquierCosa: any = 42;  
cualquierCosa = "ahora soy string"; // No da error
```

Tipos especiales en TypeScript

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Void

¿Qué es? Indica que una función NO devuelve nada (o devuelve undefined)

¿Cuándo usarlo? Para funciones que ejecutan acciones pero no retornan valores

```
typescript

function saludar(nombre: string): void {
    console.log(`Hola ${nombre}`);
    // No tiene 'return' o solo 'return;'
}

function guardarDatos(): void {
    // Guarda en base de datos pero no devuelve nada
    console.log("Datos guardados");
}
```

Tipos Compuestos/Complejos en TypeScript

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Union

¿Qué es? Permite que una variable sea de uno de varios tipos posibles

¿Cuándo usarlo? Cuando una variable puede tener diferentes tipos según el contexto

typescript

```
let id: string | number = "ABC123"; // Puede ser string o number
let estado: "loading" | "success" | "error" = "loading";
```

Tipos Compuestos/Complejos en TypeScript

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Array

¿Qué es? Una lista ordenada de elementos del mismo tipo

¿Cuándo usarlo? Para colecciones: listas de usuarios, números, tareas, etc.

typescript

```
let numeros: number[] = [1, 2, 3, 4];
let nombres: string[] = ["Ana", "Luis"];
```

Tipos Compuestos/Complejos en TypeScript

TypeScript ofrece varios tipos primitivos que son la base para definir datos en tu aplicación.

Tuple

¿Qué es? Un array con número fijo de elementos y tipos específicos en cada posición

¿Cuándo usarlo? Para coordenadas, pares clave-valor, datos estructurados fijos

typescript

```
let coordenada: [number, number] = [10, 20];
let persona: [string, number] = ["Ana", 25];
```

Funciones en TypeScript

Las funciones en TypeScript son como las de JavaScript, pero con tipos explícitos para parámetros y valor de retorno. Esto nos ayuda a:

- Detectar errores antes de ejecutar el código
- Saber exactamente qué espera y devuelve cada función
- Tener mejor autocompletado en el IDE

Declaración Básica

Función tradicional con tipos definidos para parámetros y retorno

Sintaxis

```
typescript
function nombreFuncion(parametro: tipo): tipoRetorno {
    // código
    return valor;
}
```

Ejemplo

```
typescript
// Función que suma dos números
function sumar(a: number, b: number): number {
    return a + b;
}

// Función que saluda (no devuelve nada)
function saludar(nombre: string): void {
    console.log(`Hola ${nombre}`);
}

// Función que verifica si es mayor de edad
function esMayorDeEdad(edad: number): boolean {
    return edad >= 18;
}

// Uso de las funciones
let resultado = sumar(5, 3);           // resultado: 8 (number)
saludar("Ana");                      // imprime: "Hola Ana"
let esAdulto = esMayorDeEdad(20);     // esAdulto: true (boolean)
```

Funciones en TypeScript

Parámetros Opcionales

Parámetros que pueden ser omitidos al llamar la función (se usa ?)

Sintaxis

```
typescript
function nombreFuncion(obligatorio: tipo, opcional?: tipo): tipoRetorno {
    // código
}
```

Ejemplo

```
typescript

// Función para crear usuario (apellido es opcional)
function crearUsuario(nombre: string, apellido?: string): string {
    if (apellido) {
        return `${nombre} ${apellido}`;
    }
    return nombre;
}

// Función para calcular área (altura opcional para cuadrados)
function calcularArea(ancho: number, alto?: number): number {
    if (alto) {
        return ancho * alto; // Rectángulo
    }
    return ancho * ancho; // Cuadrado
}

// Uso de las funciones
let usuario1 = crearUsuario("Juan"); // "Juan"
let usuario2 = crearUsuario("María", "González"); // "María González"

let areaCuadrado = calcularArea(5); // 25
let areaRectangulo = calcularArea(4, 6); // 24
```

Funciones en TypeScript

Parámetros Opcionales

Parámetros que pueden ser omitidos al llamar la función (se usa ?)

Sintaxis

```
typescript
function nombreFuncion(obligatorio: tipo, opcional?: tipo): tipoRetorno {
    // código
}
```

Ejemplo

```
typescript
// Función para crear usuario (apellido es opcional)
function crearUsuario(nombre: string, apellido?: string): string {
    if (apellido) {
        return `${nombre} ${apellido}`;
    }
    return nombre;
}

// Función para calcular área (altura opcional para cuadrados)
function calcularArea(ancho: number, alto?: number): number {
    if (alto) {
        return ancho * alto; // Rectángulo
    }
    return ancho * ancho; // Cuadrado
}

// Uso de las funciones
let usuario1 = crearUsuario("Juan"); // "Juan"
let usuario2 = crearUsuario("María", "González"); // "María González"

let areaCuadrado = calcularArea(5); // 25
let areaRectangulo = calcularArea(4, 6); // 24
```

¡IMPORTANTE!

```
typescript
// Los parámetros opcionales SIEMPRE van al final
function malaFuncion(opcional?: string, obligatorio: number) { } // ✗ ERROR!
function buenaFuncion(obligatorio: number, opcional?: string) { } // ✓ BIEN
```

Funciones en TypeScript

Parámetros por Defecto

Parámetros que tienen un valor predeterminado si no se proporciona

Sintaxis

typescript

```
function nombreFuncion(parametro: tipo = valorDefecto): tipoRetorno {  
    // código  
}
```

Ejemplo →

```
// Función para saludar con tratamiento por defecto  
Windsurf: Refactor | Explain | X  
function saludarFormal(nombre: string, tratamiento: string = "Sr./Sra."): string {  
    return `Hola ${tratamiento} ${nombre}`;  
}  
  
// Función para calcular precio con descuento por defecto  
Windsurf: Refactor | Explain | X  
function calcularPrecio(precioBase: number, descuento: number = 0): number {  
    return precioBase - (precioBase * descuento);  
}  
  
// Función para configurar tema de la app  
Windsurf: Refactor | Explain | X  
function configurarTema(color: string = "azul", tamaño: string = "mediano"): string {  
    return `Tema: color ${color}, tamaño ${tamaño}`;  
}  
  
// Uso de las funciones  
let saludo1 = saludarFormal("Ana");                                // "Hola Sr./Sra. Ana"  
let saludo2 = saludarFormal("Carlos", "Dr.");                         // "Hola Dr. Carlos"  
  
let precio1 = calcularPrecio(100);          // 100 (sin descuento)  
let precio2 = calcularPrecio(100, 0.15); // 85 (15% descuento)  
  
let tema1 = configurarTema();                // "Tema: color azul, tamaño mediano"  
let tema2 = configurarTema("rojo");           // "Tema: color rojo, tamaño mediano"  
let tema3 = configurarTema("verde", "grande"); // "Tema: color verde, tamaño grande"
```

Funciones en TypeScript

Funciones Flecha (Arrow Functions)

Sintaxis más concisa para escribir funciones

Sintaxis

```
typescript
// Función flecha básica
const nombreFuncion = (parametro: tipo): tipoRetorno => {
    return valor;
};

// Si es una sola linea, puedes omitir las llaves y return
const nombreFuncion = (parametro: tipo): tipoRetorno => expresion;
```

Ejemplo
→

```
// Función flecha básica
Windsurf: Refactor | Explain | X
const multiplicar = (a: number, b: number): number => {
    return a * b;
};
// Función flecha de una línea
Windsurf: Refactor | Explain | X
const duplicar = (numero: number): number => numero * 2;
// Función flecha sin parámetros
Windsurf: Refactor | Explain | X
const obtenerFechaActual = (): string => new Date().toLocaleDateString();
// Función flecha con un parámetro (paréntesis opcionales)
Windsurf: Refactor | Explain | X
const esParImpar = (numero: number): string => numero % 2 === 0 ? "par" : "impar";
// Función flecha que devuelve objeto (requiere paréntesis)
Windsurf: Refactor | Explain | X
const crearPersona = (nombre: string, edad: number): {nombre: string, edad: number} => ({
    nombre: nombre,
    edad: edad
});
// Con parámetros opcionales y por defecto
Windsurf: Refactor | Explain | X
const formatearTexto = (texto: string, mayuscula: boolean = false): string =>
    mayuscula ? texto.toUpperCase() : texto.toLowerCase();
// Uso de las funciones flecha
let producto = multiplicar(4, 5);           // 20
let doble = duplicar(7);                    // 14
let fecha = obtenerFechaActual();          // "28/7/2025" (depende de tu configuración)
let tipo = esParImpar(8);                   // "par"
let persona = crearPersona("Luis", 30);     // {nombre: "Luis", edad: 30}
let textoMinuscula = formatearTexto("HOLA"); // "hola"
let textoMayuscula = formatearTexto("hola", true); // "HOLA"
```

Funciones en TypeScript

Función Tradicional | Funciones Flecha (Arrow Functions)

```
// Función tradicional
Windsurf: Refactor | Explain | X
function sumarTradicional(a: number, b: number): number {
    return a + b;
}

// Función flecha equivalente
Windsurf: Refactor | Explain | X
const sumarFlecha = (a: number, b: number): number => a + b;

// Función tradicional con parámetros opcionales y por defecto
Windsurf: Refactor | Explain | X
function procesarDatos(datos: string, formato?: string, debug: boolean = false): string {
    if (debug) console.log("Procesando...");
    return formato ? `${formato}: ${datos}` : datos;
}

// Función flecha equivalente
Windsurf: Refactor | Explain | X
const procesarDatosFlecha = (datos: string, formato?: string, debug: boolean = false): string => {
    if (debug) console.log("Procesando...");
    return formato ? `${formato}: ${datos}` : datos;
};
```

¿PASAMOS A LA PRÁCTICA?



Preparando tu Entorno: VS Code y TypeScript

Empezar con TypeScript en VS Code es sencillo y te proporciona una excelente experiencia de desarrollo.

Instalar Node.js

TypeScript se ejecuta sobre Node.js.
Si no lo tienes, descárgalo e instálalo
desde nodejs.org.



Instalar TypeScript Globalmente

Abre tu terminal y ejecuta el siguiente comando para instalar el compilador de TypeScript (tsc).

```
npm install -g typescript
```

Verificar Instalación

Confirma que TypeScript se ha instalado correctamente.

tsc --version



Tu Primer Proyecto TypeScript en VS Code

Ahora que tienes TypeScript instalado, crea tu primer archivo y compílalo.

1

Crear Carpeta del Proyecto

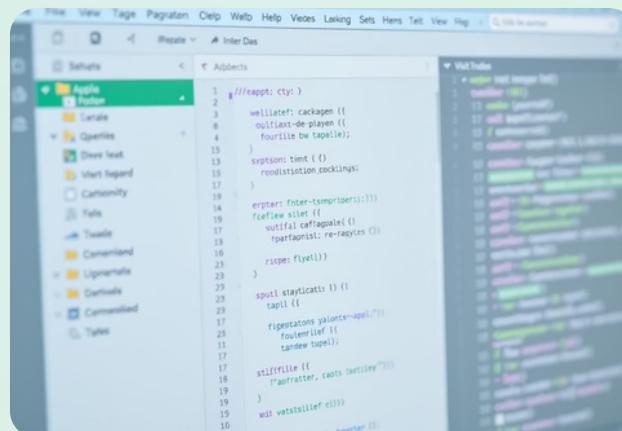
Crea una nueva carpeta para tu proyecto y ábrela en VS Code.

```
mkdir mi-ts-app  
cd mi-ts-app  
code .
```

2

Crear Archivo TypeScript

Dentro de VS Code, crea un archivo llamado app.ts.



3

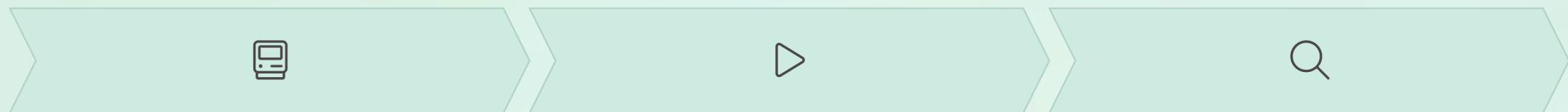
Escribir Código TypeScript

Añade el siguiente código simple a app.ts.

```
function saludar(nombre: string): string {  
    return `Hola, ${nombre}!`;  
}  
  
let usuario = "Mundo";  
  
console.log(saludar(usuario));
```

Compilando y Ejecutando tu Código TS

El compilador de TypeScript (tsc) transformará tu código .ts en JavaScript para que puedas ejecutarlo por Node.js o en el navegador.



Compilar el Archivo

En la terminal de VS Code, ejecuta el compilador. Esto creará app.js en la misma carpeta.

```
tsc app.ts
```

Ejecutar el Archivo JS

Ahora puedes ejecutar el archivo JavaScript generado usando Node.js. Verás la salida en la terminal.

```
node app.js
```

Explora el Archivo .js

Abre app.js y verás el JavaScript transpilado, ¡sin tipos!

```
// app.js
function saludar(nombre) {
  return `Hola, ${nombre}!`;
}

let usuario = "Mundo";
console.log(saludar(usuario));
```

¡GRACIAS POR VUESTRA ATENCIÓN!



RECURSOS

[Tipos básicos | Clase 1 | Curso TypeScript](#)

[Tipos de Datos Básicos en TypeScript -- Curso de TypeScript #03](#)

[Documentación oficial TypeScript](#)

[Repositorio Github Esther Tapias | Píldora TypeScript](#)