Clase 1: Fundamentos de TypeScript - Potenciando tu JavaScript

---

1. ¿Qué es TypeScript? Una Extensión Inteligente de JavaScript

TypeScript no es un lenguaje completamente nuevo, sino un **superset de JavaScript**. Esto significa que es una capa por encima de JavaScript que añade características adicionales, principalmente el **tipado estático opcional**, y luego se "compila" de nuevo a JavaScript para que pueda ser ejecutado por navegadores o Node.js.

- **Superset de JavaScript:** Cualquier código JavaScript válido es también un código TypeScript válido. Podés simplemente cambiar la extensión de un archivo `.js` a `.ts` y, en teoría, funcionará (aunque para aprovechar las ventajas de TypeScript, tendrás que empezar a añadir tipos).
- **Tipado Estático Opcional:** Esta es la joya de la corona de TypeScript. A diferencia de JavaScript, donde los tipos de las variables se determinan durante la ejecución (tipado dinámico), TypeScript te permite definir los tipos de datos en el momento de escribir el código (tiempo de desarrollo o compilación). Esto es "opcional" porque no estás obligado a tipar todo; TypeScript es lo suficientemente inteligente como para **inferir** muchos tipos por sí mismo.
- **Compilación (Transpilación) a JavaScript:** Dado que los entornos de ejecución (navegadores, Node.js) solo entienden JavaScript, tu código TypeScript (`.ts`) debe ser transformado a JavaScript (`.js`) por el compilador de TypeScript (`tsc`). Este proceso se llama **transpilación**.

¿Por qué existe TypeScript? La Evolución de JavaScript

JavaScript es increíblemente flexible, pero esa flexibilidad puede ser un arma de doble filo, especialmente en proyectos grandes. Sus principales limitaciones son:

- **Errores en Tiempo de Ejecución:** Sin tipado estático, muchos errores comunes (como intentar sumar un número a un `undefined`) solo se detectan cuando el programa ya está corriendo, lo que hace la depuración más lenta y costosa.
- **Mantenibilidad y Escalabilidad:** Es difícil mantener y escalar grandes bases de código JavaScript, ya que la ausencia de tipos puede llevar a código confuso y difícil de refactorizar para equipos grandes.
- **Experiencia del Desarrollador (DX):** Las herramientas de desarrollo (IDEs) tienen dificultades para ofrecer autocompletado y refactorización precisos sin información de tipos.

TypeScript fue creado por Microsoft para solucionar estos problemas, permitiéndonos construir aplicaciones JavaScript a gran escala con mayor **confianza, eficiencia y seguridad**.

---

2. Ventajas y Desventajas de Usar TypeScript

Adoptar TypeScript es una decisión estratégica con pros y contras.

2.1. Ventajas Clave de TypeScript

1. **Detección Temprana de Errores:** La ventaja más significativa. TypeScript captura una gran cantidad de errores de programación (errores de tipo) durante el proceso de compilación, antes de que tu código llegue a producción. Esto ahorra horas de depuración.
2. **Mayor Mantenibilidad y Legibilidad:** Los tipos actúan como una forma de **documentación viva** de tu código. Al ver los tipos, cualquiera puede entender rápidamente qué tipo de datos espera una función o qué estructura tiene un objeto, lo que facilita el mantenimiento y la comprensión del código.
3. **Productividad del Desarrollador (Developer Experience - DX):**
    - **Autocompletado Inteligente:** Tu editor de código (como VS Code) puede ofrecer sugerencias de código mucho más precisas, reduciendo la necesidad de memorizar APIs o buscar documentación externa.
    - **Refactorización Segura:** Cambiar nombres de variables, propiedades o funciones en todo tu proyecto se vuelve más seguro, ya que el compilador te alertará sobre cualquier lugar donde ese cambio pueda romper el código.
    - **Navegación del Código:** Es más fácil saltar a la definición de una función o ver todas las referencias a una variable.
4. **Escalabilidad:** TypeScript brilla en proyectos grandes y complejos, y en equipos numerosos. La claridad que aportan los tipos y la detección temprana de errores facilitan enormemente la colaboración y la gestión de la complejidad.
5. **Adopción de Características Modernas de JavaScript:** TypeScript suele integrar las últimas características del estándar ECMAScript (el estándar de JavaScript) mucho antes de que sean ampliamente compatibles con todos los navegadores o entornos de Node.js, ya que las transpilas a una versión de JavaScript que sí es compatible.

2.2. Desventajas de TypeScript

1. **Curva de Aprendizaje Inicial:** Si vienes de JavaScript puro, hay una curva de aprendizaje para entender el sistema de tipos y las nuevas características. Requiere un cambio de mentalidad.
2. **Tiempo de Compilación:** El proceso de compilación (`.ts` a `.js`) añade un pequeño tiempo extra al ciclo de desarrollo. Sin embargo, en proyectos modernos, esto suele ser gestionado eficientemente por herramientas de build.
3. **Configuración Inicial:** Configurar un proyecto TypeScript puede requerir un poco más de configuración inicial, especialmente con el archivo `tsconfig.json`.
4. **Verbosidad Potencial:** En algunos casos, el código TypeScript puede parecer un poco más "verboso" debido a las anotaciones de tipo, aunque esto se compensa con la claridad y la seguridad.

---

3. Comparativa: TypeScript vs. JavaScript

La diferencia fundamental entre TypeScript y JavaScript radica en el **tipado**.

Característica	JavaScript	TypeScript
Tipado	**Dinámico:** Los tipos se comprueban en tiempo de ejecución. Una variable puede cambiar de tipo en cualquier momento.	**Estático Opcional:** Los tipos se comprueban en tiempo de compilación. Las variables mantienen su tipo definido (o inferido).
Detección de Errores	Mayormente en tiempo de ejecución. Los errores de tipo rompen el programa.	Mayormente en tiempo de compilación. El compilador te avisa de errores antes de ejecutar.
Refactorización	Riesgosa y propensa a errores sin herramientas avanzadas.	Más segura y con un soporte robusto de IDEs.
Legibilidad y Docs.	Depende de la buena documentación y comentarios.	Mejorada por los tipos, que documentan el código de forma inherente.
Herramientas de Dev.	Autocompletado y sugerencias limitadas en IDEs.	Autocompletado, navegación y sugerencias inteligentes.
Escalabilidad	La complejidad aumenta rápidamente en proyectos grandes.	Gestiona la complejidad y la colaboración mejor en proyectos a escala.
Ejecución	Interpretado (ejecución directa por un motor JS).	Compilado a JavaScript (requiere un paso de transpilación).

---

4. Tipos Básicos en TypeScript

TypeScript extiende los tipos primitivos de JavaScript y añade otros para mayor seguridad y expresividad.

- **number**: Para números enteros y de punto flotante.
- TypeScript

```
let edad: number = 30;
let precio: number = 19.99;
// edad = "treinta"; // Error: Type 'string' is not assignable to type 'number'.
```

- **string**: Para cadenas de texto.
- TypeScript

```
let nombre: string = "Alice";
let mensaje: string = `Hola, ${nombre}`;
```

- **boolean**: Para valores verdaderos o falsos (true/false).
- TypeScript

```
let estaActivo: boolean = true;
```

- **any**: Un tipo "comodín". Desactiva la comprobación de tipos para esa variable. Úsalo con precaución, solo cuando no sabes el tipo (ej. datos de una API externa no tipada) y necesitas flexibilidad. Pierdes las ventajas de TypeScript.
- TypeScript

```
let valorDinamico: any = 10;
valorDinamico = "Puede ser un string";
valorDinamico = true; // Todo es válido con 'any'
```

- **Inferencia de Tipos**: TypeScript es lo suficientemente inteligente como para adivinar el tipo de una variable si la inicializas con un valor. No necesitas anotarla explícitamente siempre.
- TypeScript

```
let cantidad = 5; // TypeScript infiere que 'cantidad' es de tipo number
// cantidad = "cinco"; // Error, ya que 'cantidad' es number
```

---

5. Colecciones y Estructuras de Datos Básicas

Más allá de los primitivos, TypeScript nos permite tipar estructuras más complejas.

5.1. Arrays Tipados

Define que todos los elementos de un array son de un tipo específico.

- **Sintaxis `Tipo[]`**:
- TypeScript

```
let numeros: number[] = [1, 2, 3];
// numeros.push("cuatro"); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.
```

- **Sintaxis Genérica `Array<Tipo>`**:
- TypeScript

```
let nombres: Array<string> = ["Ana", "Luis", "Pedro"];
```

5.2. Tuplas (`Tuple`)

Las tuplas son arrays con un número fijo de elementos, donde el tipo de cada elemento en una posición específica es conocido.

TypeScript

```
// Define una tupla para coordenadas [longitud, latitud]
let coordenadas: [number, number] = [10.5, 20.3];
// let errorCoordenadas: [number, number] = ["diez", 20.3]; // Error: Type 'string' is not assignable to type 'number'.
// let errorLongitud: [number, number] = [10.5, 20.3, 30.0]; // Error: Source has 3 elements, but target has 2.
```

5.3. Enums (`enum`)

Las enumeraciones permiten definir un conjunto de constantes relacionadas, lo que mejora la legibilidad y la seguridad del código.

Enum Numérico (por defecto): Asigna valores numéricos.

- TypeScript

```
enum Direccion {
    Norte,  // 0
    Este,   // 1
    Sur,    // 2
    Oeste   // 3
}
let miDireccion: Direccion = Direccion.Este;
console.log(miDireccion); // Salida: 1
// Puedes obtener el nombre del miembro usando el valor numérico (reverse mapping):
console.log(Direccion[miDireccion]); // Salida: "Este"
```

Enum de Cadena (String Enum): Asigna valores de tipo `string`. Más legibles en la salida y depuración.

- TypeScript

```
enum EstadoRespuesta {
    Exito = "EXITO",
    Error = "ERROR",
    Cargando = "CARGANDO"
}
let estadoActual: EstadoRespuesta = EstadoRespuesta.Exito;
console.log(estadoActual); // Salida: "EXITO"
// Nota: Los string enums no tienen reverse mapping a diferencia de los numéricos.
```

Enum Heterogéneo (no recomendado): Mezcla números y cadenas. Generalmente se desaconseja por la falta de claridad.

---

6. Interfaces: Definiendo la Forma de los Objetos

Las **interfaces** son una de las características más potentes de TypeScript. Actúan como un **"contrato"** o un **"plano"** que define la **estructura (forma)** que debe tener un objeto. No generan código JavaScript en tiempo de ejecución; solo existen para las comprobaciones de tipo en tiempo de compilación.

- **Sintaxis:**
- TypeScript

```
interface Usuario {
    id: number;
    nombre: string;
    email: string;
    edad?: number; // La '?' hace que la propiedad sea opcional
    esActivo: boolean;
}
```

- **Creando Objetos que Cumplen la Interfaz:**
- TypeScript

```
const usuario1: Usuario = {
    id: 1,
    nombre: "Juan Perez",
    email: "juan@example.com",
    esActivo: true
};
```

```
// const usuario2: Usuario = { id: 2, nombre: "Ana" }; // Error: Property 'email' and 'esActivo' are missing.
```

-
-

6.1. Extender Interfaces

Las interfaces pueden extender otras interfaces, heredando todas sus propiedades y métodos. Esto permite construir estructuras de datos complejas de manera modular.

```
TypeScript

interface Punto {
    x: number;
    y: number;
}

interface Circulo extends Punto { // Circulo hereda x y y de Punto
    radio: number;
}

const miCirculo: Circulo = {
    x: 10,
    y: 20,
    radio: 5
};
```

También es posible **redefinir propiedades** al extender una interfaz, pero solo si la nueva definición es un **subtipo** (más específico) de la propiedad original.

```
TypeScript

interface Elemento {
    id: string; // id es un string general
}

interface Boton extends Elemento {
    // id se redefine a un subtipo de string (unión de literales)
    id: "btn-aceptar" | "btn-cancelar";
    texto: string;
}

const botonAceptar: Boton = {
    id: "btn-aceptar",
    texto: "Aceptar"
};
// const botonError: Boton = { id: "otro-id", texto: "X" }; // Error: Type '"otro-id"' is not assignable...
```

6.2. Agregando Funciones a Interfaces

Las interfaces no solo definen datos, sino también el **comportamiento** que un objeto debe tener. Puedes incluir la **firma de una función** (parámetros y tipo de retorno) en una interfaz.

TypeScript

```
interface Calculadora {
    sumar(a: number, b: number): number;
    restar(a: number, b: number): number;
}

// Un objeto que implementa la interfaz
const miCalculadora: Calculadora = {
    sumar: function(a, b) { return a + b; },
    restar: (a, b) => a - b // También se puede usar función de flecha si 'this' no es un problema
};

console.log(miCalculadora.sumar(5, 3)); // Salida: 8
```

Importante sobre this en interfaces y objetos literales:

Cuando definas un método en un objeto literal y necesites usar this para referenciar propiedades de ese mismo objeto (como this.nombre), debes usar una función regular (function() { ... } o la sintaxis abreviada metodo() { ... }). Las funciones de flecha (() => { ... }) capturan el this de su contexto léxico superior, que a menudo no es el objeto mismo en el que se definen.

---

7. Funciones Tipadas

TypeScript nos permite tipar los **parámetros** y el **valor de retorno** de una función, lo que garantiza que se usen y se comporten de manera predecible.

- **Parámetros Tipados:**
- TypeScript

```
function saludar(nombre: string): void { // 'void' indica que la función no retorna nada
    console.log(`Hola, ${nombre}`);
}
// saludar(123); // Error: Argument of type 'number' is not assignable to parameter of type 'string'.
saludar("Mundo");
```

- **Retorno Tipado:**
- TypeScript

```
function sumarNumeros(a: number, b: number): number { // Indica que retorna un 'number'
    return a + b;
```

```
}
let resultado = sumarNumeros(10, 5); // resultado es inferido como 'number'
// function dividir(a: number, b: number): string { return a / b; } // Error: Type 'number' is not assignable to type 'string'.
```

- **Funciones con Interfaces como Parámetros:**
- TypeScript

```
interface Producto {
    nombre: string;
    precio: number;
}

function mostrarDetallesProducto(producto: Producto): void {
    console.log(`Producto: ${producto.nombre}, Precio: $${producto.precio}`);
}

const miProducto = { nombre: "Laptop", precio: 1200 };
mostrarDetallesProducto(miProducto);
// mostrarDetallesProducto({ nombre: "Teclado" }); // Error: Property 'precio' is missing.
```

---

8. Paradigmas y Patrones de Diseño con TypeScript

TypeScript se integra excelentemente con varios paradigmas y patrones de diseño, amplificando sus beneficios gracias al tipado estático:

8.1. Paradigmas de Programación que Complementa

- **Programación Orientada a Objetos (POO):** TypeScript añade características fuertes de POO (Clases, Interfaces, Modificadores de Acceso como `private`, `protected`) que permiten construir sistemas con encapsulación, herencia y polimorfismo robustos.
- **Programación Funcional (PF):** El tipado de funciones, parámetros y retornos asegura la "pureza" de las funciones, facilitando la composición y la inmutabilidad, pilares de la PF.
- **Programación Basada en Componentes:** Fundamental en el desarrollo frontend moderno. TypeScript permite tipar las propiedades (props) y el estado de los componentes, garantizando la correcta comunicación y uso entre ellos.
- **Programación Genérica:** Permite escribir código flexible que funciona con una variedad de tipos de datos sin perder la seguridad del tipado, promoviendo la reutilización.

8.2. Patrones de Diseño Beneficiados por TypeScript

Los patrones de diseño son soluciones probadas a problemas comunes en el diseño de software. TypeScript no solo permite implementarlos, sino que los **refuerza** a través de su sistema de tipos:

- **Patrones de Creación (ej., Factory, Builder, Singleton):** TypeScript asegura que los objetos creados cumplan con las interfaces o tipos esperados, y que su construcción siga reglas estrictas.
- **Patrones Estructurales (ej., Adapter, Decorator, Facade):** Las interfaces y los tipos en TypeScript definen claramente los contratos de las interacciones y las composiciones, haciendo estas estructuras más comprensibles y seguras.
- **Patrones de Comportamiento (ej., Observer, Strategy, Command):** TypeScript garantiza que la comunicación entre objetos sea tipada y coherente, lo que simplifica la gestión de flujos de trabajo y comportamientos dinámicos.