🚀 Módulo 1: Fundamentos de TypeScript (8:30 - 9:30)

Hora Aproximada	Contenido	Tópicos Clave
8:30 - 9:30	Módulo 1: Fundamentos de TypeScript	* **¿Qué es TypeScript?** (Ventajas sobre JavaScript, el compilador tsc).
		* **Configuración del Entorno** (Node.js, npm, tsconfig.json).

¿Qué es TypeScript? 🤔

TypeScript es un **superset de JavaScript** que compila a JavaScript plano. La característica principal es el **tipado estático opcional**.

- **Ventajas Clave sobre JavaScript:**
  - **Tipado Estático:** Permite detectar errores comunes en tiempo de compilación (antes de que el código se ejecute), lo que lleva a un código más robusto y menos propenso a errores en producción.
  - **Mejor Legibilidad y Mantenimiento:** El tipado actúa como documentación explícita.
  - **Herramientas Potentes (IntelliSense):** Los IDEs (como VS Code) pueden ofrecer autocompletado y refactorización mucho más avanzados gracias a la información de tipo.
  - **Soporte para Características Modernas:** Incluye soporte para características de ECMAScript futuras que aún no están disponibles en todos los entornos.

- **El Compilador tsc (TypeScript Compiler):** Es la herramienta que toma el código .ts y lo transforma en código .js ejecutable. **El Navegador nunca ejecuta TypeScript directamente.**

Configuración del Entorno 🛠️

1. **Requisitos:** Asegúrate de tener **Node.js y npm** instalados (son el estándar para el desarrollo moderno de JavaScript/TypeScript).
2. **Inicialización del Proyecto:**
   Bash
   `npm init -y`
   `npm install typescript --save-dev`

3. **Configuración del Compilador (tsconfig.json):** Este archivo es el **corazón de un proyecto TypeScript**. Define cómo debe comportarse el compilador tsc.
   - Se genera con el comando npx tsc --init.
   - **Propiedades Clave a Entender:**
     - target: La versión de JavaScript a la que se compilará el código (ej: "es2020").
     - module: El sistema de módulos a usar (ej: "commonjs" o "esnext").
     - rootDir y outDir: Dónde están los archivos .ts y dónde poner los archivos .js compilados.
     - strict: La bandera más importante; debe estar en true para las mejores prácticas.

🧱 Módulo 2: Tipos Básicos (9:30 - 10:30)

Hora Aproximada	Contenido	Tópicos Clave
9:30 - 10:30	Módulo 2: Tipos Básicos	* **Tipos Primitivos:** string, number, boolean.
		* **Tipado Implícito vs. Explícito.**

Tipos Primitivos 🥇

Son los bloques de construcción de cualquier código.

- **number**: Representa tanto números enteros como de punto flotante. **No hay distinción** entre int y float como en otros lenguajes.
  TypeScript
  ```
  let edad: number = 30;
  let pi: number = 3.14159;
  ```

- **string**: Para datos textuales. Permite el uso de comillas simples (') dobles (") o *template literals* (`).
  TypeScript
  ```
  let nombre: string = "Alicia";
  let saludo: string = `Hola, mi nombre es ${nombre}.`;
  ```

- **boolean**: Solo puede ser true o false.
  TypeScript
  ```
  let esActivo: boolean = true;
  ```

- **Otros Tipos Primitivos (Mención Rápida):** bigint (para números muy grandes) y symbol.

Tipado Implícito vs. Explícito ✍️

Este es un concepto fundamental para entender cómo funciona el inferenciador de tipos de TypeScript.

- **Tipado Explícito:** Le dices explícitamente a TypeScript qué tipo de dato tendrá una variable usando la sintaxis : Tipo.
  TypeScript
  ```
  let miNumero: number = 10; // Explícito: sabes que es un número
  ```
  - **Mejor Práctica:** Útil para parámetros de funciones y cuando inicializas una variable con null o undefined.
- **Tipado Implícito (Inferencia de Tipos):** TypeScript es lo suficientemente inteligente como para *inferir* (adivinar) el tipo de una variable si la inicializas inmediatamente.
  TypeScript
  ```
  let miCadena = "Hola Mundo"; // Implícito: TypeScript infiere que es de tipo 'string'
  // miCadena = 42; // ¡Error!
  ```
  - **Mejor Práctica:** A menudo es la forma más limpia, ya que el tipado se infiere correctamente y reduce la redundancia.

✨ Módulo 3: Tipos Especiales (10:30 - 11:00)

Hora Aproximada	Contenido	Tópicos Clave

10:30 - 11:00	Módulo 3: Tipos Especiales	* any, unknown, void, null y undefined, never.
		* **Aserciones de Tipo** (as y <>).

El Espectro de Tipos Especiales 👻

- **any (⚠️ Evitar):** Desactiva el chequeo de tipos para esa variable. Es la "puerta de escape" de TypeScript, permitiendo que la variable sea reasignada a cualquier cosa.
  TypeScript
  ```
  let data: any = "Hola";
  data = 10; // Válido con 'any', pero pierde la seguridad de tipos.
  ```

- **unknown (✅ Preferir):** Un tipo más seguro que any. Una variable de tipo unknown puede contener cualquier valor, **PERO** no se puede realizar ninguna operación con él hasta que se haya *probado* su tipo (se requiere un *type check* o una aserción).
  TypeScript
  ```
  let valor: unknown = "un texto";
  // valor.toUpperCase(); // Error!
  if (typeof valor === 'string') {
    valor.toUpperCase(); // Válido dentro del if
  }
  ```

- **void:** Se usa para indicar que una función **no devuelve ningún valor** (o devuelve undefined).
  TypeScript
  ```
  function logMessage(msg: string): void {
    console.log(msg);
  ```

```
}
```

- **null y undefined:** Representan la ausencia de valor. Por defecto, son subtipos de todos los tipos a menos que la opción strictNullChecks esté activa (lo cual es una buena práctica).
  - null: Ausencia de valor con intención.
  - undefined: Valor que aún no ha sido asignado.
- **never:** Representa un valor que **nunca ocurrirá**. Se usa típicamente para funciones que **lanzan una excepción** o que entran en un **bucle infinito** inalcanzable.

```TypeScript
function error(mensaje: string): never {
    throw new Error(mensaje);
}
```

Aserciones de Tipo (Type Assertions) ⚖️

Le dices al compilador: **"Confía en mí, sé más que tú sobre el tipo de esta variable."** No realizan comprobaciones en tiempo de ejecución.

- **Sintaxis con as (Preferida en React/TSX):**

```TypeScript
let codigo: any = "12345";
let largo: number = (codigo as string).length;
```

- **Sintaxis con <> (Estilo Antiguo/No funciona en TSX):**

```TypeScript
let codigo2: any = "67890";
let largo2: number = (<string>codigo2).length;
```

- **Advertencia:** Úsalas con moderación, ya que eliminan la seguridad del tipado. Es mejor depender de la inferencia de tipos o las comprobaciones de tipo (instanceof, typeof).

⚙️ Módulo 4: Práctica y Taller (11:00 - 11:30)

Hora Aproximada	Contenido	Tópicos Clave
11:00 - 11:30	Práctica y Taller	*** Configuración de un proyecto inicial.**
		*** Ejercicios de tipado básico y uso de any vs. unknown.**

Taller Práctico 👩‍💻

1. **Configuración Rápida:**
   - Crear una carpeta de proyecto y generar el tsconfig.json.
   - Crear un archivo index.ts y escribir un simple programa **"Hola Mundo"**.
   - Compilar con tsc y ejecutar el .js resultante con node.
2. **Ejercicios de Tipado Estricto:**
   - Declarar variables usando tipado explícito e implícito con string, number, y boolean.
   - Crear una función que reciba dos números y devuelva un number (suma).

- Crear una función que no devuelva nada (usando void).
- **El Desafío any vs. unknown:**
  - Crear una función que acepte un argumento de tipo any y llame a un método que no existe (ej: arg.noExiste()). *Resultado: Sin error en compilación, falla en tiempo de ejecución.*
  - Repetir la función con tipo unknown. *Resultado: Error de compilación, forzando al desarrollador a realizar una comprobación de tipo antes de usarlo (ej: if (typeof arg === 'object') { ... }), demostrando la seguridad.*

🛠️ Práctica y Taller: Tipado Básico y Seguridad (11:00 - 11:30)

1. Configuración de un Proyecto Inicial

El objetivo es tener un entorno de desarrollo funcional para ejecutar código TypeScript.

Paso 1: Inicialización y Dependencias

Abre tu terminal y ejecuta los siguientes comandos:

```
Bash
```

```
# 1. Crear y entrar a la carpeta del proyecto
mkdir ts-dia1-taller
cd ts-dia1-taller

# 2. Inicializar un proyecto Node.js
npm init -y

# 3. Instalar TypeScript como dependencia de desarrollo
npm install typescript --save-dev

# 4. Instalar ts-node (para ejecutar archivos TS directamente sin compilarlos primero)
npm install ts-node --save-dev
```

Paso 2: Generar el tsconfig.json

Este es el archivo de configuración clave. Usaremos la opción --init y luego modificaremos la configuración para asegurar el **tipado estricto**.

Bash

```
# Generar el archivo de configuración
npx tsc --init
```

Modificación Clave en tsconfig.json:

Asegúrate de que estas opciones estén configuradas para las mejores prácticas:

- "target": "es2020": Para compilar a una versión moderna de JavaScript.
- "module": "commonjs": El sistema de módulos estándar para Node.js.
- "strict": true: **La configuración más importante.** Activa todas las comprobaciones estrictas de tipado.

Paso 3: Crear el Archivo de Trabajo

Crea un archivo llamado index.ts y añade un código simple para probar la configuración.

index.ts

TypeScript

```
const curso: string = "TypeScript Fundamental";
const horas: number = 4;

function bienvenida(nombre: string, tema: string): void {
  console.log(`Hola, ${nombre}! Bienvenido al curso de ${tema}.`);
}

bienvenida("Estudiante Proactivo", curso);
console.log(`Duración de hoy: ${horas} horas.`);
```

Paso 4: Ejecución

En lugar de compilar con tsc (que crea un archivo .js), usaremos ts-node para ejecutar el archivo .ts directamente.

Bash

```
npx ts-node index.ts
```

Salida esperada:

> Hola, Estudiante Proactivo! Bienvenido al curso de TypeScript Fundamental.
> Duración de hoy: 4 horas.

---

2. Ejercicios de Tipado y any vs. unknown

El objetivo es experimentar la **seguridad** que ofrece TypeScript y contrastarla con la falta de seguridad de any.

Ejercicio A: El Peligro de any (Tipo Inseguro)

Declara una variable con tipo any y observa cómo el compilador no detecta un error obvio.

index.ts (Añadir al final)

TypeScript

```
console.log("\n--- Ejercicio A: Any ---");

let datosSinControl: any = 100;

// ⚠️ TypeScript lo permite, ya que 'any' desactiva la comprobación.
// Esto FALLARÁ en tiempo de ejecución (runtime error).
// Un número no tiene el método toUpperCase.
try {
    const resultado = datosSinControl.toUpperCase();
    console.log("Resultado: ", resultado);
} catch (error) {
    console.error("Error capturado en tiempo de ejecución (¡peligro!):", error.message);
}

// Ahora lo reasignamos a otro tipo
datosSinControl = "typescript";
console.log("Valor reasignado y usado: ", datosSinControl.toUpperCase());
```

Ejercicio B: La Seguridad de unknown (Tipo Seguro)

Declara la misma variable como unknown y observa cómo TypeScript te obliga a comprobar el tipo.

index.ts (Añadir al final)

TypeScript

```
console.log("\n--- Ejercicio B: Unknown ---");

let datosDesconocidos: unknown = 100;

// ❌ Intenta descomentar la línea de abajo. El compilador lanzará un error.
// datosDesconocidos.toUpperCase();

// ✅ Para usar métodos, TypeScript nos fuerza a hacer una Aserción o Comprobación de Tipo.
if (typeof datosDesconocidos === 'string') {
    // Dentro de este bloque, TypeScript sabe que es un string y permite usar sus métodos.
    console.log("Es un string, OK: ", datosDesconocidos.toUpperCase());
} else if (typeof datosDesconocidos === 'number') {
    // Dentro de este bloque, TypeScript sabe que es un number.
    console.log("Es un número, OK: ", datosDesconocidos.toFixed(2));
} else {
    console.log("Tipo no manejado.");
}
```

Punto Clave para la Clase:

Conclusión: any te da libertad, pero a costa de la seguridad (errores en producción). unknown te da seguridad al forzarte a manejar todos los posibles tipos **antes** de realizar operaciones, evitando *runtime errors*. **Siempre prefiere unknown sobre any**.
