UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN III
FRONT END

Unidad Temática N° 3:
Typescript

Teórico

2° Año – 3° Cuatrimestre

Índice

Introducción a TypeScript

TypeScript es un lenguaje de programación desarrollado por Microsoft en 2012. Es un **superconjunto de JavaScript** (superset) que agrega **tipado estático** y otras funcionalidades avanzadas al lenguaje JavaScript, facilitando el desarrollo de aplicaciones a gran escala de manera más robusta y mantenible. En pocas palabras, todo programa válido de JavaScript también lo es en TypeScript, pero además podemos añadirle tipos y características adicionales.

Es importante destacar que **TypeScript no se ejecuta directamente** en el navegador ni en Node.js. El código TypeScript primero debe *transpilarse* (compilarse) a JavaScript puro. Esta tarea la realiza el **compilador de TypeScript** (`tsc`), generando archivos `.js` equivalentes que sí pueden ejecutarse en cualquier entorno JavaScript.

Algunas razones clave para usar TypeScript:

- **Menos errores en ejecución:** Gracias al tipado estático, muchos errores de tipo se detectan en tiempo de compilación (antes de ejecutar el código). Por ejemplo, TypeScript evitará que sumemos inadvertidamente un número con una cadena, mostrando un error antes de que el código corra. En cierto modo, TypeScript actúa como un **guardián**: si intentas asignar o pasar un tipo incorrecto, te avisará inmediatamente, reduciendo la probabilidad de fallos durante la ejecución.
- **Mejor autocompletado y documentación:** Los editores de código (VS Code, WebStorm, etc.) aprovechan TypeScript para brindar **autocompletado inteligente** y detección temprana de errores. Esto acelera el desarrollo y mejora la **productividad**, ya que el IDE puede sugerir propiedades y métodos válidos, y advertirnos si usamos algo incorrectamente.
- **Mantenibilidad y escalabilidad:** Al definir explícitamente tipos y estructuras (por ejemplo, con *interfaces*), es más fácil para varios desarrolladores trabajar en un proyecto grande sin malentendidos. TypeScript ayuda a que el código sea más **legible y auto-documentado**, facilitando su mantenimiento a largo plazo.

A pesar de estas ventajas, TypeScript **no reemplaza** a JavaScript en tiempo de ejecución; más bien actúa como una capa de desarrollo. Tras compilar, el resultado final es JavaScript estándar. Esto significa que podemos adoptar TypeScript gradualmente en un proyecto JavaScript existente, sabiendo que al final del proceso todo seguirá siendo JS.

Compilación a JavaScript: Para convertir archivos TypeScript a JavaScript utilizamos el comando de compilación de TypeScript. Por ejemplo, asumiendo que tenemos un archivo hola.ts, podemos compilarlo desde la terminal:

```
tsc hola.ts
```

Si el compilador está correctamente instalado, este comando producirá un archivo `hola.js` con el contenido traducido a JavaScript. Si nuestro código TypeScript tiene errores de tipos, veremos mensajes de error en la consola durante la compilación; aun así (por defecto) el compilador generará el archivo `.js`. Si deseamos ser más estrictos y evitar emitir código JavaScript cuando hay errores, podemos usar la opción `--noEmitOnError`:

```
tsc --noEmitOnError hola.ts
```

Con esta opción, si existen errores de compilación, **no** se generará ningún archivo de salida.

Para instalar TypeScript, se puede usar el gestor de paquetes **npm**. Por ejemplo, para una instalación global podríamos ejecutar:

```
npm install -g typescript
```

Esto instala el compilador a nivel global, habilitando el comando `tsc` en la terminal. Alternativamente, en un proyecto específico podemos agregar TypeScript como dependencia de desarrollo.

TypeScript encaja perfectamente con el ecosistema JavaScript moderno. En las siguientes secciones exploraremos sus principales características y cómo difiere de JavaScript, con ejemplos sencillos que ilustran cada concepto.

Tipos Básicos

Tipos primitivos

TypeScript incluye los mismos **tipos primitivos** básicos de JavaScript:

- **string** – representa texto (cadenas de caracteres). Ejemplo: "`Hello, world`".
- **number** – representa tanto números enteros como de punto flotante. En JavaScript (y TypeScript) *todos* los números se manejan con el mismo tipo primitivo `number` (no existe una distinción separada entre enteros y flotantes).
- **boolean** – representa valores lógicos `true` o `false`.

Estos tipos funcionan igual que en JavaScript. Podemos usar operadores y funciones estándar sobre ellos. Por ejemplo:

```
let mensaje: string = "Hola, TypeScript";
let cantidad: number = 42;
let encendido: boolean = true;
```

Arreglos (Arrays)

Para definir un **arreglo** (array) de elementos de un tipo específico, TypeScript ofrece dos sintaxis equivalentes. Podemos usar `Tipo[]` (corchetes) después del nombre del tipo, o la forma genérica `Array<Tipo>`:

```
let numeros: number[] = [1, 2, 3];          // Array de números
let palabras: Array<string> = ["uno", "dos", "tres"]; // Array de strings
```

En el primer caso `numeros` está declarado como "arreglo de número", y en el segundo `palabras` como `Array` de `string`. Ambas notaciones son intercambiables.

El tipo any

TypeScript tiene un tipo especial llamado `any`, que efectivamente desactiva el chequeo de tipos para ese valor. Usar `any` significa que le estamos diciendo al compilador "no compruebes estrictamente este valor; confía en mí". Esto puede ser útil al migrar código JavaScript existente o cuando un valor dinámico es difícil de tipar.

Cuando una variable es de tipo `any`, podemos hacer prácticamente *cualquier* operación sobre ella sin que TypeScript marque error de compilación:

```
let desconocido: any = { x: 0 };
desconocido.foo();          // llamar un método (aunque no exista)
desconocido();              // invocarlo como función
desconocido.bar = 100;      // asignar nuevas propiedades
desconocido = "ahora soy una cadena";  // cambiar completamente el tipo del valor
const n: number = desconocido;         // asignarlo a una variable de tipo estricto
```

Todas las líneas anteriores son "válidas" en TypeScript si `desconocido` es `any`, aunque probablemente muchas causarían errores en tiempo de ejecución si se ejecutasen. **Se recomienda evitar** el uso de `any` siempre que sea posible, ya que al usarlo perdemos muchas de las ventajas de detección de errores que ofrece TypeScript. Es mejor usar `any` solo como último recurso o en casos muy puntuales.

Anotaciones de tipo e inferencia

En TypeScript podemos **anotar** explícitamente el tipo de una variable al declararla, usando la sintaxis `: Tipo` después del nombre de la variable. Por ejemplo:

let miNombre: string = "Juan";

Aquí le indicamos al compilador que `miNombre` *debe* ser de tipo `string`. Sin embargo, en muchos casos esto no es necesario. TypeScript cuenta con **inferencia de tipos**: si una variable se inicializa con un valor, el compilador automáticamente deduce el tipo en base a ese valor inicial.

Por ejemplo, declarar:

```
let miCiudad = "Buenos Aires";
```

hará que TypeScript infiera que `miCiudad` es de tipo `string` (porque su valor inicial es una cadena). Si luego intentáramos asignar un número a `miCiudad`, TypeScript reportaría un error de tipo, a pesar de que no declaramos explícitamente el tipo, ya que infiere el tipo a partir del valor inicial.

En general, conviene aprovechar la inferencia de tipos de TypeScript para reducir redundancia. Solo necesitaremos **anotaciones de tipo** explícitas cuando el contexto no provea suficiente información al compilador o para mejorar la legibilidad del código.

Funciones

Las **funciones** en TypeScript (igual que en JavaScript) son bloques reutilizables de código que pueden recibir parámetros y retornar un valor. TypeScript extiende la sintaxis de funciones de JavaScript permitiendo **anotar los tipos** de los parámetros y del valor de retorno, lo que ayuda a prevenir llamados incorrectos.

Parámetros de función

Podemos especificar el tipo de cada parámetro en la firma de la función, colocando `:Tipo` después del nombre del parámetro. Por ejemplo:

```
function saludar(nombre: string) {
  console.log("Hola, " + nombre.toUpperCase() + "!!");
}
```

En este caso, la función `saludar` espera un parámetro `nombre` de tipo `string`. Si intentamos llamar a `saludar` con un tipo distinto, TypeScript emitirá un error de compilación:

```
saludar(42);
// Error: Argument of type 'number' is not assignable to parameter of type 'string'.
```

Así nos aseguramos en tiempo de compilación de que `saludar` siempre reciba un string, evitando posibles errores en tiempo de ejecución (por ejemplo, si pasáramos un número, el método `.toUpperCase()` no existiría y JavaScript lanzaría un error).

Nota: Incluso si no anotamos explícitamente los tipos de los parámetros, TypeScript seguirá verificando el número de argumentos. Si una función espera, por ejemplo, 2 parámetros y le pasamos solo 1 al invocarla, el compilador nos advertirá que faltan argumentos.

Tipo de retorno de función

De forma similar a los parámetros, podemos anotar el **tipo de retorno** esperado de una función, colocando `:Tipo` justo después del paréntesis de parámetros. Por ejemplo:

```
function obtenerNumeroFavorito(): number {
  return 26;
}
```

Aquí indicamos que `obtenerNumeroFavorito` debe devolver un `number`. Dentro de la función, el compilador exigirá que el `return` (o los `return` en plural, si hubiera múltiples) produzcan un número. Si intentáramos retornar otro tipo (o olvidáramos retornar algo), TypeScript lo señalaría como error.

Sin embargo, en la mayoría de casos **no es necesario especificar el tipo de retorno** manualmente, porque TypeScript lo **infiere automáticamente** a partir de las expresiones de `return`. En el ejemplo anterior, aunque no hubiéramos escrito `:number`, el compilador deduciría que la función devuelve un número porque retornamos 26. Añadir la anotación de retorno explícita puede ser útil como documentación o para forzar un tipo deseado, pero si no la ponemos, TypeScript hará una inferencia inteligente.

Cabe mencionar que si una función **no devuelve ningún valor** (es decir, no tiene `return` o solo `return` sin valor), su tipo de retorno es void. Podemos anotarlo explícitamente o dejar que TypeScript lo infiera. Por ejemplo, una función que solo imprime un mensaje podría definirse como `function imprimirMensaje(msg: string): void { ... }`. El uso de `void` indica que la función no produce un resultado aprovechable, lo cual también es verificado por el compilador.

Tipos Avanzados (Objetos, Alias, Interfaces)

Además de los tipos primitivos, TypeScript ofrece herramientas para describir tipos más complejos y crear **tipos personalizados**. A continuación, exploraremos varios conceptos avanzados: cómo definir tipos de **objetos** con propiedades, combinar tipos con **uniones**, crear **alias** y **interfaces** para estructurar tipos, manejar valores opcionales, y usar **enumeraciones** para conjuntos de constantes.

Tipos de objetos

En JavaScript, muchos valores son objetos con propiedades. TypeScript nos permite describir la **forma** de esos objetos indicando qué propiedades tienen y de qué tipo es cada una. Podemos definir un tipo de objeto **en línea** usando una sintaxis de literal de objeto con los nombres de propiedad y sus tipos:

```
function imprimirCoordenadas(pt: { x: number; y: number }) {
  console.log("Coordenada X:", pt.x);
  console.log("Coordenada Y:", pt.y);
}

imprimirCoordenadas({ x: 3, y: 7 });
```

En este ejemplo, la función `imprimirCoordenadas` espera como parámetro un objeto que debe tener dos propiedades: `x` y `y`, ambas de tipo `number`. Al llamar a la función, pasamos efectivamente un objeto con esa estructura `{ x: 3, y: 7 }`. Si intentáramos llamar `imprimirCoordenadas({ x: 3 })` o con propiedades de otro tipo, TypeScript reportaría un error porque el argumento no cumple con el tipo requerido.

Propiedades opcionales

Al definir objetos, es posible que algunas propiedades no siempre estén presentes. Podemos marcar propiedades como **opcionales** utilizando un signo de interrogación `?` después del nombre de la propiedad en la definición de tipo. Por ejemplo:

```
function imprimirNombre(obj: { nombre: string; apellido?: string }) {
  console.log("Nombre:", obj.nombre);
  if (obj.apellido !== undefined) {
    console.log("Apellido:", obj.apellido);
  }
}

imprimirNombre({ nombre: "Bob" });                          // Apellido opcional omitido
imprimirNombre({ nombre: "Alice", apellido: "Alisson" }); // Apellido proporcionado
```

Aquí la función `imprimirNombre` acepta un objeto con una propiedad obligatoria `nombre` (string) y una propiedad opcional `apellido` (string). Gracias a `apellido?: string`, podemos llamar a la función tanto con el apellido como sin él. Dentro de la función, usualmente tendremos que **comprobar si la propiedad opcional existe** (`!== undefined`) antes de usarla, para evitar acceder a un valor `undefined`.

Tipos de unión

Un **tipo de unión** permite indicar que un valor puede ser de **más de un tipo**. Se denota usando el símbolo `|` entre dos o más tipos. Por ejemplo, `number | string` es un tipo que indica un valor que puede ser o bien número o bien cadena.

Las uniones son útiles para modelar situaciones en las que un valor puede ser de distintos tipos. Por ejemplo:

```
function imprimirId(id: number | string) {
  console.log("ID:", id);
}
```

La función `imprimirId` acepta un parámetro que puede ser o un número o una cadena. Podemos llamarla tanto con `imprimirId(101)` como con `imprimirId("ABC")`. Sin embargo, trabajar con uniones dentro de la función requiere cierta precaución. Dentro de `imprimirId`, el parámetro `id` *podría* ser cualquiera de los dos tipos.

TypeScript solo nos permitirá realizar operaciones con `id` que sean válidas para **ambos** posibles tipos. Si intentáramos usar un método específico de cadena, como `id.toUpperCase()`, el compilador se quejará, porque si `id` fuera un número, esa operación no existe:

```
function imprimirId(id: number | string) {
  // Error si intentamos: id.toUpperCase();
  if (typeof id === "string") {
    console.log(id.toUpperCase());  // En este bloque, id es string
  } else {
    console.log(id);                // En este bloque, id es number
  }
}
```

En el código anterior, utilizamos una condición `typeof` para **refinar** el tipo de `id`. Dentro del bloque `if`, TypeScript reconoce que `id` es un `string` (porque esa rama solo corre cuando el tipo es string), por lo que permite usar `toUpperCase()`. En el bloque `else`, `id` solo puede ser numérico, así que simplemente lo imprimimos.

Este proceso se conoce como *narrowing* (acotación del tipo) y es cómo TypeScript maneja las uniones de forma segura: debemos verificar explícitamente el tipo en tiempo de ejecución para poder usar las operaciones específicas de ese tipo.

Si todos los miembros de una unión comparten alguna propiedad o método en común, entonces podemos usarlo sin necesidad de refinar. Por ejemplo, tanto los strings como los arrays tienen la propiedad `length` y el método `slice`. Así que podríamos escribir:

```
function primerosTres(x: number[] | string) {
  return x.slice(0, 3);
}
```

En este caso, `x.slice(0, 3)` es válido porque independientemente de si x es un array de números o una cadena, ambos tipos tienen definido `slice`. El tipo de retorno aquí será inferido como `number[] | string` (según qué camino ocurra en tiempo de ejecución).

Alias de tipo

Un **alias de tipo** nos permite crear un nombre personalizado para un tipo ya existente (sea un tipo primitivo, unión u objeto complejo). Se define con la palabra clave `type`. Por ejemplo, podríamos querer dar un nombre descriptivo a una unión de tipos:

```
type Identificador = number | string;
```

Ahora `Identificador` actúa como un tipo que abarca tanto números como cadenas. Podemos usarlo así: `let userId: Identificador = 123;` o `userId = "ABC";`.

Los alias también son útiles para nombrar tipos de objetos complejos, haciendo el código más legible. Por ejemplo:

```
type CarYear = number;
type CarModel = string;
type CarType = string;

type Car = {
  year: CarYear;
  model: CarModel;
  type: CarType;
};

const myCar: Car = {
  year: 2021,
  model: "Corolla",
  type: "Toyota"
};
```

Aquí definimos alias para representar componentes de un auto (`CarYear`, `CarModel`, `CarType`) y luego un alias `Car` que es un objeto con esas propiedades. Al final, usamos `Car` para tipar la variable `myCar`. Esto mejora la claridad: cualquiera puede ver que `myCar` es de tipo `Car`, y luego remitirse a la definición de `Car` para ver sus propiedades.

Interfaces

Las **interfaces** en TypeScript son otra forma de nombrar y definir la forma de un objeto. Una interfaz declara un conjunto de propiedades y sus tipos, y los objetos que la implementen deben cumplir esa estructura. Por ejemplo, la interfaz equivalente a nuestro último ejemplo podría ser:

```
interface Car {
  year: number;
  model: string;
  type: string;
}

function mostrarCar(car: Car) {
  console.log(`Auto ${car.type} ${car.model}, Año ${car.year}`);
}
```

Esta interfaz `Car` describe el mismo tipo de objeto con propiedades `year`, `model` y `type` todas obligatorias. La función `mostrarCar` espera un objeto que cumpla con la interfaz `Car`. Podemos llamarla con cualquier objeto que tenga esa estructura (ya sea que provenga de una variable declarada como `Car` o un literal con esas propiedades).

Interfaces y tipos de objeto *anónimos* (definidos en línea como en ejemplos anteriores) son muy similares: ambos definen formas de objetos. La diferencia es que la interfaz le da un **nombre reutilizable** a ese tipo de objeto, lo que puede ser útil para referenciarlo en múltiples lugares o para establecer contratos más complejos.

Por ejemplo, podríamos escribir otra función que acepte un `Car` o incluso hacer que una clase "implemente" la interfaz `Car` (como veremos en la sección de POO). Todo objeto que tenga las propiedades requeridas por `Car` será compatible con esta interfaz gracias al **sistema de tipos estructural** de TypeScript (basta con que la estructura coincida, no importa cómo se originó el tipo).

Diferencias entre alias de tipo e interfaces

Las **type aliases** y las **interfaces** son muy parecidas: ambas permiten definir tipos de objeto con nombres personalizados. En muchos casos se pueden usar indistintamente. Sin embargo, existen algunas diferencias sutiles:

- **Extensibilidad:** Las interfaces pueden **extenderse** (`extends`) para crear nuevas interfaces basadas en las existentes, e incluso pueden combinarse declarando el mismo nombre de interfaz en distintos lugares (TypeScript las fusiona automáticamente). En cambio, un alias de tipo se define de una vez y **no se puede volver a abrir o extender** posteriormente. Si necesitas agregar propiedades a un tipo existente, con una interfaz puedes hacerlo (creando una subinterfaz o aprovechando la fusión de declaraciones); mientras que con un alias tendrías que crear un alias nuevo (lo cual no modifica el original).
- **Tipos que pueden definir:** Las interfaces solo pueden describir la forma de objetos (incluyendo la firma de funciones y la estructura de clases), mientras que los alias de tipo son más generales: un alias puede representar un objeto, pero también un tipo primitivo, una unión, una tupla, una función, etc. Por ejemplo, no puedes crear una interfaz que sea "string o number", pero sí un `type Alias = string | number`.

En la práctica, para definir estructuras de objetos complejas en TypeScript, es cuestión de preferencia usar alias o interfaces. Muchas veces las interfaces se emplean para contratos orientados a objetos y los alias para uniones o tipos más simples, pero ambos enfoques funcionan. Lo importante es conocer que si requieres las capacidades de extensión y fusión, la interfaz te las brinda, mientras que el alias no.

Comparativa resumida:

Característica	Interface	Type Alias
Extensión (extends)	Sí, se puede extender o incluso fusionar (múltiples declaraciones con el mismo nombre se unen).	No, no se puede extender ni modificar tras la definición.
Tipos que puede definir	Principalmente tipos de objeto (estructura de propiedades/métodos, firmas de funciones, etc.).	Cualquier tipo: primitivos, uniones, tuplas, funciones, objetos, etc.
Implementación en clases	Las clases pueden implementar (implements) interfaces para asegurar que cumplen su contrato.	Las clases no pueden implementar un alias directamente (se recomienda usar interfaz en ese caso).
Sintaxis	`interface Persona {edad:number;}`	`type Persona = {edad:number};`

null y undefined

JavaScript tiene dos valores primitivos que se utilizan para señalar un valor ausente o no inicializado: `null` y `undefined`. En TypeScript, existen los tipos correspondientes null y undefined. Sin embargo, el comportamiento de estos tipos depende de la configuración del compilador.

Por defecto (es decir, con la opción `strictNullChecks` habilitada dentro del modo estricto), `null` y `undefined` **no forman parte de los demás tipos**. Esto significa que una variable de tipo `string`, por ejemplo, no podría recibir un `null` a menos que explícitamente lo incluyamos en su tipo (usando una unión, p. ej. `string | null`). Lo mismo aplica para `undefined`.

Si intentamos hacer algo como:

```
let titulo: string = "Informe";
// ... más tarde en el código:
titulo = undefined; // ❌ Error en TypeScript (strictNullChecks habilitado)
```

El compilador reclamará porque `undefined` no es asignable a un `string`. La solución sería declarar `let titulo: string | undefined` si realmente necesitamos que pueda ser indefinido en algún momento, o asegurarnos de que la variable siempre tenga un string válido.

En modos menos estrictos (desactivando `strictNullChecks`), TypeScript permite asignar `null` y `undefined` libremente a cualquier variable (emulando el comportamiento de JavaScript puro, donde todas las variables podrían potencialmente ser `undefined`). No obstante, es **recomendable usar el modo estricto** y manejar estos valores de forma explícita, ya que ayuda a prevenir errores comunes (por ejemplo, acceder a propiedades de algo que puede ser null).

En resumen, `null` y `undefined` son valores que requieren atención especial en TypeScript: conviene declararlos en el tipo cuando un valor puede faltar (p. ej. propiedad opcional o variable que comienza sin definir), o chequear su presencia antes de usarlos, para mantener la seguridad de tipos.

Enumeraciones (enum)

Una **enumeración** (declarada con la palabra clave enum) permite definir un conjunto de constantes con nombre. Las enumeraciones facilitan agrupar valores relacionados, como las direcciones cardinales, los días de la semana, etc., otorgándoles nombres descriptivos en lugar de usar números o strings "mágicos" directamente en el código.

TypeScript admite enumeraciones **numéricas** (por defecto) y **de cadenas**.

Enumeración numérica: En una enum numérica, los miembros reciben valores numéricos (incrementales por defecto). Por ejemplo:

```
enum Direccion {
  Arriba = 1,
  Abajo,
  Izquierda,
  Derecha
}
```

En este ejemplo, hemos definido una enum `Direccion` con cuatro opciones. Por defecto:

- `Direccion.Arriba` vale `1` (porque se lo asignamos explícitamente).
- `Direccion.Abajo` vale `2` (un número más que Arriba, asignado automáticamente).
- `Direccion.Izquierda` vale `3` .
- `Direccion.Derecha` vale `4` .

Si no especificamos ningún número inicial, la numeración comienza automáticamente en 0. Por ejemplo:

```
enum Respuesta {
  No,
  Si
}
```

Aquí `Respuesta.No` sería `0` y `Respuesta.Si` sería `1`. Podemos concluir que las enums numéricas asignan automáticamente valores incrementales a sus miembros cuando no se proporcionan.

Enumeración de cadenas: En una enum de cadenas, cada miembro deberá ser inicializado con una constante de texto, ya que no existe un auto-incremento lógico para strings. Ejemplo:

```
enum DireccionTexto {
  Arriba = "Arriba",
  Abajo = "Abajo",
  Izquierda = "Izquierda",
  Derecha = "Derecha"
}
```

En este caso, `DireccionTexto.Arriba` es la cadena `"Arriba"`, `DireccionTexto.Abajo` es `"Abajo"`, etc. Las enums de cadena no proporcionan un comportamiento de incremento automático; cada valor debe definirse manualmente, pero a cambio hacen claro e inmutable el valor exacto de cada opción.

Las enumeraciones, sean numéricas o de texto, se usan de forma similar:

```
let movimiento: Direccion = Direccion.Izquierda;
if (movimiento === Direccion.Izquierda) {
  console.log("El jugador se mueve a la izquierda");
}
```

Comparar valores de la enum es seguro y legible. En tiempo de ejecución, las enums **numéricas** generan un objeto que mapea tanto de nombre a valor como de valor a nombre (permitiendo ciertas operaciones de reflexión, como `Direccion[2]` devolviendo `"Abajo"`). En cambio, las enums de **cadenas** simplemente se traducen a un objeto que mapea nombre a valor (no hay mapeo inverso automático, ya que los valores de cadena no son únicos en cuanto a tipo).

Comparativa de enumeraciones numéricas vs de cadenas:

Aspecto	Enums Numéricas	Enums de Cadenas
Asignación de valores	Automática si no se especifican (inicia en 0 o en el valor dado al primer miembro, luego incrementa).	Debe asignarse manualmente un literal de string a cada miembro.
Tipo de valor	Valores numéricos (ej. 0, 1, 2, ...).	Valores tipo string (ej. "Arriba", "Abajo", ...).
Bidireccionalidad	Sí, hay mapeo inverso: dado un valor numérico se puede obtener el nombre del miembro (`Direccion[2] → "Abajo"`).	No hay mapeo inverso automático: solo se obtiene el valor a partir del nombre (no el nombre desde el valor).
Mezcla de tipos	Es posible (enums *heterogéneas* con números y strings en la misma enum), pero no es recomendable.	No aplica (todos los miembros son cadenas, mezclar perdería el sentido).

En general, las **enums numéricas** se usan cuando importan más los valores subyacentes (por ejemplo, flags bit a bit, valores integrales, etc.), mientras que las **enums de cadenas** son útiles para valores descriptivos legibles (por ejemplo, estados, acciones, etc.). Ambas sirven para dar semántica y seguridad de tipo a conjuntos de constantes lógicas en nuestros programas.

Clases y Programación Orientada a Objetos

JavaScript incorporó el concepto de **clases** en ECMAScript 2015, y TypeScript construye sobre esto agregando capacidades de tipado estático a las clases. Las clases permiten definir **modelos u objetos** con propiedades y métodos, siguiendo los principios de la Programación Orientada a Objetos (POO) como **encapsulación**, **herencia** y **polimorfismo**. Veamos cómo se declaran clases en TypeScript y qué añadidos aporta.

Definición de una clase y propiedades

En TypeScript, definimos una clase usando la palabra clave `class` seguida del nombre de la clase. Dentro podemos declarar **propiedades** (también llamadas *campos*) y métodos. Por ejemplo:

```
class Punto {
  x: number;
  y: number;

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

Esta clase `Punto` tiene dos propiedades (`x` e `y`) de tipo `number`, y un **constructor** que inicializa esas propiedades. Observa la sintaxis: anotamos los tipos de `x` e `y` tanto en la declaración de las propiedades como en los parámetros del constructor. Podemos crear instancias así:

```
const p = new Punto(10, 20);
console.log(p.x); // 10
```

Modificadores de acceso

Por defecto, todas las propiedades y métodos de una clase en TypeScript son **públicos** (`public`). Esto significa que pueden accederse desde fuera de la clase libremente. Podemos restringir el acceso usando **modificadores** de acceso:

- `public`: (valor por omisión) accesible desde cualquier lugar.
- `private`: la propiedad o método solo es accesible dentro de la propia clase.
- `protected`: accesible dentro de la clase y también por las clases hijas (subclases) que la extiendan.

Por ejemplo:

```
class Persona {
  private edad: number;
  nombre: string;  // (por omisión es pública)

  constructor(nombre: string, edad: number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  public saludar() {
    console.log(`Hola, soy ${this.nombre}`);
  }
}

const juan = new Persona("Juan", 30);
juan.saludar();          // OK, saludar es público
console.log(juan.nombre); // OK, nombre es público
// console.log(juan.edad);  // ❌ Error: edad es privada
```

En este ejemplo, intentamos acceder a `juan.edad` desde fuera de la clase Persona. Al ser `edad` privada, TypeScript emitirá un error de compilación indicando que no se puede acceder a ella. Este encapsulamiento mejora la seguridad y claridad del código, asegurando qué datos pueden ser vistos o modificados desde fuera.

TypeScript también soporta el modificador `readonly` para propiedades de clase. Si marcamos una propiedad como `readonly`, solo podrá asignársele un valor una vez (generalmente en el constructor o al declararla). Intentar cambiarla posteriormente produce un error de compilación:

```
class Circulo {
  readonly PI: number = 3.1416;
  radio: number;

  constructor(radio: number) {
    this.radio = radio;
  }

  calcularCircunferencia(): number {
    return 2 * this.PI * this.radio;
  }
}

const c = new Circulo(5);
// c.PI = 3.15; // ❌ Error: PI es de solo lectura
```

En el ejemplo, `PI` es una constante de la clase Circulo. La marcamos readonly porque su valor no debería cambiar una vez inicializado.

Nota: En el modo estricto de TypeScript, se aplica la regla de **strict property initialization**. Esto significa que todas las propiedades de instancia declaradas deben inicializarse en el constructor (o al declararlas, como hicimos con `PI`) antes de que termine la construcción del objeto. Si una propiedad no tiene valor inicial y no la asignamos en el constructor, TypeScript dará un error. Por ejemplo, si en la clase `Punto` omitiéramos la asignación de `this.x` o `this.y` en el constructor, el compilador nos recordaría que esas propiedades podrían quedar indefinidas. La forma de resolverlo es inicializarlas, marcarlas como opcionales (`x?: number`) o usar `!` (non-null assertion) si estamos seguros de que se asignarán antes de usarlas.

Además de propiedades de instancia, podemos declarar propiedades y métodos **estáticos** usando la palabra clave `static`. Los miembros estáticos pertenecen a la clase en sí, no a instancias particulares. Por ejemplo, podríamos tener `Math.PI` como un valor estático (como de hecho existe en JavaScript). Accederemos a miembros estáticos mediante el nombre de la clase (`MiClase.propiedadEstatica`) en lugar de a través de una instancia.

Constructores

El **constructor** es un método especial para inicializar nuevas instancias de la clase. En TypeScript, el constructor puede tener parámetros con tipos y también valores por defecto. Como vimos, dentro del constructor usualmente asignamos los valores recibidos a las propiedades de la clase.

Podemos aprovechar valores por defecto para simplificar la creación de objetos. Por ejemplo, podríamos permitir crear un `Punto` en el origen si no se especifican coordenadas:

```
class Punto3D {
  x: number;
  y: number;
  z: number;

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

const origen = new Punto3D();
console.log(origen.x, origen.y, origen.z); // 0 0 0
```

Aquí, el constructor de `Punto3D` asigna 0 por defecto a `x`, `y` y `z` si no se pasan argumentos al instanciar. Así, `new Punto3D()` crea un punto en (0,0,0), mientras que new `Punto3D(5, 10, 2)` crearía un punto en (5,10,2).

Una diferencia con las funciones normales es que los constructores **no tienen tipo de retorno anotado** (ni falta que hace): siempre retornan implícitamente una instancia de la clase en la que están definidos.

Métodos de instancia

Los métodos de una clase se declaran como funciones dentro de la misma. Pueden utilizar las mismas características de tipado que funciones normales (anotaciones de tipos para parámetros y retorno). Dentro de un método, para referirnos a las propiedades o a otros métodos de la instancia actual, usamos `this` (igual que en JavaScript).

Por ejemplo, añadamos un método a la clase `Punto` para escalar (multiplicar) sus coordenadas:

```ts
class Punto {
  x: number;
  y: number;
  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
  escalar(factor: number): void {
    this.x *= factor;
    this.y *= factor;
  }
}

const p = new Punto(2, 3);
p.escalar(5);
console.log(p); // Punto { x: 10, y: 15 }
```

El método `escalar` toma un número (`factor`) y no retorna nada (`void`). Simplemente multiplica las propiedades `x` e `y` de la instancia por ese factor. Nótese que es obligatorio usar `this.x` en lugar de `x` a secas para acceder a la propiedad de la instancia.

Getters y setters

TypeScript permite definir métodos especiales *getters* y *setters* para acceder y establecer propiedades calculadas o controlar el acceso a las mismas. Se declaran usando la sintaxis `get nombre()` y `set nombre(valor)` dentro de la clase.

Los getters y setters funcionan igual que en JavaScript, pero TypeScript añade verificación de tipos a su uso. Además, el compilador aplica algunas reglas:

- Si definimos solo un getter para una propiedad y no un setter, la propiedad se considera automáticamente de solo lectura (no asignable fuera de la clase).
- Si no especificamos el tipo del parámetro de un setter, TypeScript infiere que debe ser el mismo que el tipo de retorno del getter correspondiente.

Un ejemplo sencillo:

```
class Cuadrado {
  private _lado: number;
  constructor(lado: number) {
    this._lado = lado;
  }
  get lado(): number {
    return this._lado;
  }
  set lado(valor: number) {
    if (valor <= 0) {
      throw new Error("El lado debe ser positivo");
    }
    this._lado = valor;
  }
  get area(): number {
    return this._lado * this._lado;
  }
}

const sq = new Cuadrado(5);
console.log(sq.lado); // 5 (usa getter)
console.log(sq.area); // 25 (getter calculado)
sq.lado = 10;         // usa setter
// sq.lado = -3;      // ❌ Error: arrojaría excepción en tiempo de ejecución
```

En la clase `Cuadrado`, la propiedad `lado` se controla con un getter y setter:

- El getter simplemente devuelve el valor interno `_lado`.
- El setter verifica que el nuevo valor sea positivo antes de asignarlo.

También tenemos un getter calculado `area` que obtiene el área en base al lado actual (no definimos setter para `area`, por lo que `area` es de solo lectura).

Al usar `sq.lado` en el código, realmente estamos invocando el getter (y similar para asignación con el setter), pero TypeScript nos permite usar la sintaxis de propiedad normal. El chequeo de tipos asegura que a `sq.lado` solo se le asigne un número en este caso (si intentáramos `sq.lado = "abc"` daría error de compilación). La lógica del setter garantiza en tiempo de ejecución que no establezcamos valores inválidos.

Herencia (extends)

La **herencia** permite crear clases nuevas basadas en clases existentes. En TypeScript (y JS), usamos la sintaxis `extends` para indicar que una clase **hereda** de otra. La subclase (clase hija) obtiene automáticamente las propiedades y métodos de la superclase (clase padre), y puede agregar nuevos o sobreescribir algunos.

Por ejemplo:

```
class Animal {
  nombre: string;
  constructor(nombre: string) {
    this.nombre = nombre;
  }
  mover() {
    console.log(`${this.nombre} se mueve`);
  }
}

class Perro extends Animal {
  ladrar() {
    console.log("¡Guau!");
  }
}

const rocky = new Perro("Rocky");
rocky.mover();  // "Rocky se mueve"  (método heredado de Animal)
rocky.ladrar(); // "¡Guau!"         (método propio de Perro)
```

Aquí definimos una clase base `Animal` con una propiedad `nombre` y un método `mover()`. Luego definimos `Perro` que *extiende* de `Animal`. Automáticamente, `Perro` tiene la propiedad `nombre` y el método `mover()` gracias a la herencia. Añadimos además el método `ladrar()` solo para Perro. Cuando creamos un Perro con nombre "Rocky", podemos llamar `rocky.mover()` (definido en Animal) y `rocky.ladrar()` (definido en Perro).

La herencia promueve la reutilización de código y permite modelar relaciones jerárquicas. Algunos detalles a tener en cuenta:

- Si la subclase necesita un **constructor**, debe llamar al constructor de la superclase usando `super(...)` antes de usar `this`. En nuestro ejemplo, `Perro` no define constructor, pero hereda el de `Animal`. Si quisiéramos un constructor en Perro, tendría que invocar `super(nombre)` para que `Animal` inicialice `this.nombre`.

- Una subclase puede **sobrescribir** un método de la superclase para cambiar su comportamiento. TypeScript requerirá que la firma (nombre, parámetros y tipo de retorno) coincida. Podemos opcionalmente usar la palabra clave `override` al sobrescribir para dejarlo claro, pero no es obligatoria (solo es una verificación extra en tiempo de compilación).

Implementación de interfaces

Otra faceta de la POO es el concepto de interfaces como "contratos". En TypeScript, las clases pueden **implementar** interfaces utilizando la palabra clave `implements`. Esto obliga a que la clase provea las propiedades o métodos que la interfaz describe.

Por ejemplo, supongamos una interfaz sencilla:

```
interface Saludador {
  saludar(nombre: string): void;
}
```

Esta interfaz dice que cualquier entidad "Saludador" debe tener un método `saludar` que reciba un string y no retorne nada. Podemos hacer que una clase la implemente:

```
class Robot implements Saludador {
  saludar(nombre: string): void {
    console.log(`Hola ${nombre}, soy un robot.`);
  }
}
```

Al marcar `class Robot implements Saludador`, TypeScript verifica que `Robot` efectivamente tenga un método `saludar(nombre: string): void`. Si lo omitimos o si la firma no coincide, el compilador dará un error, obligándonos a cumplir el contrato.

Una clase puede implementar múltiples interfaces separadas por comas, lo que permite cierto grado de "herencia múltiple" de comportamientos. También una clase puede extender otra clase *y* a la vez implementar interfaces adicionales.

Con estas características —propiedades con tipos, modificadores de acceso, herencia de clases e implementación de interfaces— TypeScript nos da un sistema potente para programación orientada a objetos, similar al de lenguajes como Java o C#, pero manteniendo la flexibilidad de JavaScript.

Módulos

Cuando un programa crece, es importante poder dividir el código en archivos y **módulos** para organizarlo. TypeScript soporta el sistema de módulos de ECMAScript (ES6). Un **módulo** es simplemente un archivo que *exporta* algo (variables, funciones, clases, tipos) o *importa* algo. Cada archivo con al menos una declaración `import` o `export` se considera un módulo. En cambio, un archivo sin esas declaraciones se trata como un script global (sus variables y funciones quedan en el ámbito global).

La idea central de los módulos es que el código de un archivo está aislado del de otro a menos que explícitamente exportemos/importemos lo necesario. Esto ayuda a evitar colisiones de nombres y a mantener el código modular y limpio.

Exportando elementos

Para hacer disponible una variable, función, clase u otro elemento fuera de un archivo, lo **exportamos**. Hay dos formas principales de exportar:

- **Export nombrado:** Podemos preceder cualquier declaración con la palabra clave export para exportarla. También podemos exportar varias cosas juntas al final del archivo.
- **Export por defecto:** Cada módulo puede marcar opcionalmente **una** exportación como *default*, que sería la exportación principal por omisión.

Ejemplo de exportaciones nombradas:

```
// archivo matematicas.ts
export const PI = 3.14;
export function cuadrado(x: number): number {
  return x * x;
}

// También podríamos hacer exportaciones agrupadas:
const PHI = 1.61;
export { PHI };  // exportar constante ya declarada
```

Aquí, `matematicas.ts` exporta una constante `PI` y una función `cuadrado`. También exporta la constante PHI usando una sintaxis agrupada al final.

Ejemplo de exportación por defecto:

```
// archivo saludo.ts
export default function saludar() {
  console.log("Hola, mundo!");
}
```

En este caso, el módulo exporta una función como su valor por defecto. Notemos que tras `export default` no es necesario un nombre de función (podríamos ponerlo, pero el importante es el valor en sí). Un módulo solo puede tener una exportación default, pero puede tener múltiples exportaciones nombradas adicionales.

Importando elementos

Para usar código exportado desde otro módulo, utilizamos **importaciones**. Siguiendo los ejemplos anteriores:

```
// archivo main.ts
import { PI, cuadrado, PHI } from "./matematicas.js";
import saludar from "./saludo.js";

console.log("PI =", PI);
console.log("PHI =", PHI);
console.log("5 al cuadrado =", cuadrado(5));
saludar();
```

En el import de `matematicas.js`, usamos llaves `{ }` para listar las exportaciones específicas que queremos (deben coincidir con los nombres exportados). En el import de `saludo.js`, como ese módulo tiene una exportación por defecto, lo importamos dándole un nombre local (`saludar` en este caso) sin llaves.

Variantes de sintaxis de import:

- **Importar con alias:** Podemos cambiar el nombre de algo al importarlo, útil para evitar conflictos o por claridad:

```
import { PI as PiConstant } from "./matematicas.js";
console.log(PiConstant);
```

Aquí importamos `PI` pero lo referiremos internamente como `PiConstant`.

- **Importar todo el módulo:** Podemos importar todas las exportaciones en un solo objeto usando `* as <alias>`:

```
import * as math from "./matematicas.js";
console.log(math.PI);
console.log(math.cuadrado(3));
```

Esto reúne todas las exportaciones de `matematicas.js` bajo el objeto namespace `math`.

- **Exportar desde módulos externos:** Un módulo puede re-exportar elementos de otro módulo. Por ejemplo, `export * from "./otraFuente.js";` en un módulo A tomará todo lo exportado de "otraFuente" y lo exportará a su vez desde el módulo A.
- **Combinando default y nombrados:** Podemos importar una exportación por defecto junto con otras nombradas:

```
import defaultFunc, { otraCosa } from "./modulo.js";
```

Donde `defaultFunc` corresponderá a la exportación default y otraCosa a una exportación específica.

Exportando e importando tipos

En TypeScript, los tipos (por ejemplo, interfaces, alias de tipo, etc.) no existen en tiempo de ejecución, pero igual podemos exportarlos e importarlos para usarlos durante la fase de compilación. La sintaxis es la misma:

```
// archivo modelos.ts
export interface Usuario {
  nombre: string;
  edad: number;
}
export type ID = number | string;
```

```
// archivo app.ts
import { Usuario, ID } from "./modelos.js";

const u: Usuario = { nombre: "Ana", edad: 25 };
let identificador: ID;
identificador = 123;
identificador = "ABC";
```

Aquí exportamos una interfaz `Usuario` y un alias `ID` desde "modelos.ts". En "app.ts", los importamos con los mismos nombres. Esto permite compartir definiciones de tipo entre archivos, garantizando consistencia en todo el proyecto.

Nota: Al compilar a JavaScript, las importaciones y exportaciones de tipos se **eliminan** (no generan código), ya que solo son relevantes en tiempo de desarrollo para el verificador de tipos. Sin embargo, es una buena práctica exportar tipos junto con la lógica correspondiente para mantener el código organizado y comprensible.

En resumen, los módulos en TypeScript funcionan igual que en JavaScript moderno, pero con el beneficio adicional de poder manejar también nuestras definiciones de tipos. Usando módulos, podemos dividir un proyecto en piezas más pequeñas, controlar qué se expone públicamente y qué queda encapsulado, y mantener un código más limpio y mantenible.

Decoradores

Los **decoradores** son una característica experimental de TypeScript inspirada en lenguajes como Python o Java (anotaciones). Un decorador es básicamente una función especial que se puede aplicar a **clases**, **métodos**, **propiedades** o **parámetros** para alterar o extender su comportamiento de forma declarativa. La sintaxis es usar la `@` seguida del nombre del decorador justo encima de la declaración que queremos decorar.

Por ejemplo, podríamos tener un decorador de clase que registre algo cuando la clase se define:

```
function MostrarFechaCreacion(ctor: Function) {
  console.log(`Fecha de creación: ${new Date().toLocaleDateString()}`);
}

@MostrarFechaCreacion
class Persona {
  constructor(public nombre: string, public edad: number) {}
}
```

Al aplicar `@MostrarFechaCreacion` antes de la clase `Persona`, estamos indicando que se ejecute la función `MostrarFechaCreacion` pasándole la clase como argumento. En este caso, cuando se carga este código, el decorador imprimirá la fecha actual en la consola. Si ejecutáramos algo como `new Persona("Juan", 30)`, veríamos en consola algo como "Fecha de creación: 22/4/2025" *antes* de cualquier otra salida relacionada con Persona.

Tipos de decoradores

Los decoradores pueden utilizarse para muchas cosas, como registrar metadatos, alterar definiciones (por ejemplo, añadir propiedades o modificar métodos), o aplicar patrones generales (logging, validación, etc.) sin ensuciar la lógica principal. TypeScript permite crear distintos tipos de decoradores:

- **Decoradores de clase:** funciones que reciben el constructor de la clase decorada.
- **Decoradores de método:** reciben el prototipo, el nombre del método y su descriptor, permitiendo modificar el método o su metadata.
- **Decoradores de accesor (getter/setter):** similares a los de método, pero aplicados a propiedades con getters/setters.
- **Decoradores de propiedad:** reciben el prototipo y el nombre de la propiedad (no el valor, ya que no se aplica en tiempo de ejecución directamente).
- **Decoradores de parámetro:** reciben la función (o método) y el índice del parámetro, permitiendo asociar metadata a parámetros específicos.

Esta versatilidad viene con la advertencia de que los decoradores aún son **experimentales**. Para utilizarlos, debemos habilitar la opción experimentalDecorators en nuestro `tsconfig.json`. Por ejemplo:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    // ... otras opciones ...
  }
}
```

Con esta opción activada, el compilador TypeScript reconocerá la sintaxis de decoradores. Es importante destacar que actualmente los decoradores no forman parte del estándar JavaScript final (están en proceso de estandarización), pero TypeScript nos permite usarlos adelantándonos a su posible incorporación.

En general, los decoradores proporcionan una forma poderosa de meta-programación en TypeScript, aunque conviene usarlos con moderación y entender bien su funcionamiento. Dado que agregan lógica "oculta" detrás de declaraciones, se deben documentar y probar cuidadosamente. A pesar de ser avanzados, pueden mejorar la legibilidad en ciertos contextos (por ejemplo, frameworks como Angular hacen uso intensivo de decoradores para declarar componentes, inyectables, etc., de manera concisa).

tsconfig.json y Configuración del Proyecto

A medida que un proyecto TypeScript crece, resulta conveniente tener un lugar para definir la configuración de compilación en vez de escribir opciones en la línea de comando cada vez. El archivo **tsconfig.json** es ese lugar: un archivo JSON que indica al compilador de TypeScript cómo debe comportarse. En él podemos especificar las opciones de compilación (versión de JS de salida, directorios de entrada/salida, verificaciones estrictas, etc.), así como qué archivos incluir o excluir del proceso.

¿Qué ventajas ofrece `tsconfig.json`?

- **Centralización de configuración:** En lugar de recordar flags de compilación, todo el equipo puede referirse a este archivo. Al compartir el proyecto, otros desarrolladores solo necesitan correr `tsc` y obtendrán el comportamiento definido en `tsconfig.json`.
- **Facilita el trabajo en equipo:** Todos usarán las mismas opciones de compilación, evitando el clásico "a mí me funciona" debido a configuraciones distintas.
- **Escalabilidad:** Permite personalizar el compilador para proyectos grandes. Por ejemplo, habilitar comprobaciones estrictas puede prevenir errores conforme el proyecto crece, o ajustar la salida para compatibilidad con distintos entornos.
- **Integración con herramientas:** Muchos entornos (IDEs, bundlers como Webpack, etc.) detectan tsconfig.json y lo usan para integrar TypeScript automáticamente en flujos de trabajo.

Para inicializar rápidamente un proyecto TypeScript con un tsconfig.json básico, podemos ejecutar el comando:

```
tsc --init
```

Esto creará un archivo `tsconfig.json` con valores predeterminados que luego podemos editar según nuestras necesidades.

Veamos algunas de las opciones más comunes dentro de `tsconfig.json` (todas van dentro del objeto `"compilerOptions"`):

- `target`: Especifica la versión de JavaScript a la que se traducirá el código TypeScript. Ejemplos: `"ES5"`, `"ES6"/"ES2015"`, `"ESNext"`. Por ejemplo, si

usamos `target: "ES5"`, TypeScript emitirá código compatible con navegadores antiguos (ES5).

- `module`: Establece el sistema de módulos de salida. Ejemplos: `"commonjs"` (para Node.js), `"esnext"` o `"ES2020"` (para módulos ECMAScript nativos), `"AMD"`, etc. Depende de dónde va a correr el código compilado.
- `strict`: Si vale `true`, habilita **todas** las comprobaciones estrictas de TypeScript (es un *conjunto* que incluye `noImplicitAny`, `strictNullChecks`, `strictFunctionTypes`, etc.). Es muy recomendable activarlo para tener la máxima seguridad de tipos.
- `noImplicitAny`: Cuando está en `true`, TypeScript marcará como error cualquier variable cuyo tipo no pueda inferir (y que por tanto sería `any` implícito). Obliga a ser explícito en casos ambiguos.
- `outDir`: Directorio de salida donde colocar los archivos JavaScript compilados. Por ejemplo, `"outDir": "./dist"` haría que todos los `.js` salgan en la carpeta `dist`.
- `rootDir`: Directorio raíz del código fuente TypeScript. Sirve para que el compilador mantenga la misma estructura de directorios al emitir los archivos en outDir.
- `esModuleInterop`: Si está en `true`, habilita la inter-operación cómoda con módulos CommonJS que exportan a través de `module.exports`. Esta opción permite importarlos con la sintaxis de ES Module (`import x from "..."`) sin problemas.
- `skipLibCheck`: En `true`, omite la verificación de tipos de los archivos de declaración (`*.d.ts`) de las bibliotecas externas. Esto puede acelerar la compilación, evitando revisar definiciones de bibliotecas que generalmente ya son confiables.
- `sourceMap`: Si se habilita (`true`), genera archivos `.map` junto con los `.js` compilados. Estos archivos fuente-mapa sirven para depurar: permiten que las herramientas de desarrollo muestren el código TypeScript original al poner breakpoints, en lugar del código JavaScript transpilado.
- `noEmitOnError`: Cuando se activa, le indica al compilador que **no genere** archivos JavaScript si ocurrió algún error de compilación. Por defecto, TypeScript compila incluso si hay errores (muestra los errores, pero igual

produce los `.js`). Con esta opción en `true`, evitamos obtener archivos de salida si el código no pasó todas las comprobaciones.

Además de `compilerOptions`, el `tsconfig.json` puede tener campos como:

- `include`: Un arreglo de patrones (globs) o archivos específicos que se *incluirán* en la compilación. Ejemplo: `"include": ["src/**/*.ts", "tests/**/*.ts"]` para incluir todos los `.ts` en `src` y `tests`.
- `exclude`: Lista de patrones de archivos a *excluir*. Por defecto, si no se especifica, TypeScript excluye node_modules. Podemos añadir otros, como carpetas de build (ej. "exclude": ["dist", "out"]).

Con un `tsconfig.json` bien configurado, compilar es tan sencillo como ejecutar tsc en la raíz del proyecto (o usar un script de npm como `npm run build` si lo definimos). El compilador leerá este archivo y aplicará todas las opciones al generar el resultado. Esto hace más reproducible y sencilla la gestión del proyecto TypeScript.

Comparación entre JavaScript y TypeScript

Dado que TypeScript se basa en JavaScript, comparten gran parte de la sintaxis y semántica. Sin embargo, existen diferencias clave que es importante tener claras:

Aspecto	JavaScript (JS)	TypeScript (TS)
Tipado de variables	Dinámico (no hay tipos estáticos; el tipo de una variable puede cambiar en runtime).	Estático (se pueden declarar tipos para variables, parámetros, etc., y el compilador verifica consistencia). Los tipos son opcionales en TS, pero si se usan, deben cumplirse.
Detección de errores	En tiempo de **ejecución** principalmente. Los errores de tipo o referencia aparecen cuando el código se ejecuta (salvo que usemos herramientas como linters o tests para detectarlos antes).	En tiempo de **compilación**. Muchos errores se detectan al transpilar el código, antes de ejecutarlo, reduciendo fallos en producción. (TS no elimina la necesidad de pruebas, pero atrapa una gran clase de errores temprano).
Ejecución	Se ejecuta directamente en un motor de JS (navegador, Node.js) sin paso de compilación previo.	No se puede ejecutar tal cual; necesita transpilarse a JavaScript puro mediante el compilador TS. El resultado final es JS, que sí se ejecuta en el motor correspondiente.

Sintaxis extra	Solo soporta la sintaxis estándar de ECMAScript (según la versión usada). No tiene noción nativa de interfaces, tipos genéricos, decoradores, sobrecarga de funciones, etc. (aunque mediante patrones o librerías se pueden lograr funcionalidades similares).	Añade sintaxis y características que no existen en JS: anotaciones de tipo (: Tipo), interfaces, genéricos, enums, decoradores, modificadores de acceso en clases, sobrecarga de funciones, entre otros. Estas características se eliminan o transforman al compilar a JS (sirven en desarrollo, pero no hay equivalentes directos en runtime salvo el código que TS genere).
Compatibilidad	Es el lenguaje nativo de la web; cualquier entorno que entienda JS puede ejecutar el código.	Es un *superconjunto* de JS: cualquier código JavaScript válido es también válido en TypeScript. Podemos usar bibliotecas JS en TS mediante definiciones de tipos (.d.ts). No obstante, el código TS debe convertirse a JS para poder ejecutarse.
Flujo de trabajo	No requiere fase de compilación, lo que simplifica empezar rápidamente, pero en proyectos grandes puede dificultar el mantenimiento (por la ausencia de tipos).	Requiere una fase de compilación/transpilación (integrada fácilmente con herramientas como npm scripts, Webpack, etc.). Esto agrega un paso extra, pero mejora la robustez del código. Los editores suelen integrar la compilación/diagnóstico TS en vivo, ofreciendo una experiencia de desarrollo fluida.

En resumen, **TypeScript mejora JavaScript** incorporando el tipado estático y otras facilidades del mundo de los lenguajes compilados, a costa de un paso de compilación. TypeScript está pensado para hacer más seguro y escalable el desarrollo, especialmente en aplicaciones complejas o equipos grandes, mientras que JavaScript puro sigue siendo flexible y apropiado para scripts pequeños o para quienes prefieren la rapidez sin configuración. Afortunadamente, no es una elección excluyente de uno u otro: podemos empezar con JavaScript e ir añadiendo TypeScript gradualmente, beneficiándonos de lo mejor de ambos.

Administración del Proyecto con npm

En el ecosistema Node.js (y en el desarrollo frontend moderno), **npm** (Node Package Manager) es la herramienta estándar para gestionar las dependencias y comandos de un proyecto. Cuando iniciamos un proyecto TypeScript (especialmente si va a ejecutarse con Node o si usamos paquetes de terceros), es común inicializar un archivo `package.json`.

El archivo `package.json` sirve como **configuración principal** del proyecto. Contiene metadatos y una lista de dependencias, entre otras cosas. Algunos elementos que define:

- Información básica del proyecto (nombre, versión, autor, licencia, etc.).
- Dependencias necesarias en producción (qué paquetes requiere nuestra aplicación para funcionar).
- Dependencias de desarrollo (paquetes necesarios solo durante la fase de desarrollo o compilación, por ejemplo TypeScript en sí, frameworks de testing, herramientas de bundling, etc.).
- Scripts de npm (tareas que podemos ejecutar con `npm run`, como build, test, start, etc.).
- Punto de entrada principal (`main`) de la aplicación (archivo que representa la entrada del programa, relevante sobre todo en proyectos Node o en paquetes para publicar).

¿Por qué es importante usar package.json?

- **Gestión de dependencias:** Nos permite declarar qué librerías usamos (y qué versiones). Con solo este archivo, otra persona (o nosotros mismos en otra máquina) puede ejecutar `npm install` y obtener todas las dependencias necesarias instaladas automáticamente.
- **Automatización mediante scripts:** Podemos definir comandos en la sección "scripts" (ej.: `"build": "tsc"` o `"start": "node dist/index.js"`) y luego ejecutarlos con `npm run build`, `npm run start`, etc. Esto facilita tareas comunes sin memorizar largos comandos.
- **Facilita la colaboración:** Estandariza la forma de construir, probar y ejecutar el proyecto. Cualquier colaborador sabe que `npm test` ejecutará las pruebas si así lo definimos, o que `npm run build` compilará el proyecto, ya que esas tareas están descritas en package.json.
- **Control de versiones de paquetes:** Al listar las dependencias con números de versión, garantizamos que todos usen versiones compatibles. npm puede bloquear (lock) versiones para mayor reproducibilidad (ver archivo `package-lock.json`).

A continuación, un breve repaso a campos típicos de un `package.json`:

- `name`: Nombre del proyecto o paquete.

version: Versión actual del proyecto (siguiendo típicamente SemVer, ej. "1.0.0").

- `scripts`: Objeto que define comandos personalizados que puedes correr con `npm run <nombre>`. Por ejemplo: `"start": "node dist/index.js"` o `"build": "tsc"`. Luego `npm run start` ejecutará Node sobre el archivo indicado, y npm run build ejecutará el compilador TypeScript.
- `dependencies`: Paquetes requeridos en **producción** (objeto con nombres de paquetes y versiones). Ej.: `"express": "^4.17.1"` indica que usamos Express versión 4.x. Estas dependencias se instalan con `npm install`.
- `devDependencies`: Paquetes necesarios solo para **desarrollo** (no para la ejecución final). Ej.: `"typescript": "^5.0.0"`. Incluye tipicamente al propio TypeScript, herramientas de testing, linters, etc. Estas se instalan con `npm install --save-dev`.
- `main`: Archivo principal del proyecto (por ejemplo `"main": "dist/index.js"` una vez compilado). Sirve para señalar el punto de entrada de la aplicación o el fichero que exporta la librería en caso de publicar un paquete.
- `license`: Licencia del proyecto (ej. `"MIT"`, `"GPL-3.0"`). Importante si se distribuye públicamente, para indicar bajo qué términos se puede usar el código.

(Existen otros campos adicionales según el caso de uso, pero los anteriores son los más comunes.)

Comandos útiles con npm y TypeScript:

- **Iniciar un proyecto npm:** `npm init -y` – Crea un `package.json` con valores por defecto (-y auto-responde "yes" a todas las preguntas).
- **Instalar TypeScript (localmente en el proyecto):** `npm install typescript` --save-dev – Agrega TypeScript como dependencia de desarrollo del proyecto. Esto instala el compilador `tsc` dentro de `node_modules`.

- **Inicializar tsconfig:** `npx tsc --init` – Crea un archivo `tsconfig.json` con configuración base (usamos `npx` para ejecutar el `tsc` local sin instalarlo globalmente).
- **Compilar el proyecto:** `npx tsc` – Ejecuta el compilador TypeScript según las opciones de `tsconfig.json`. (Equivalente a ejecutar `tsc` si lo tenemos globalmente o definido en un script npm).
- **Compilación continua:** `npx tsc -w` – Inicia el modo *watch*, que recompila automáticamente al detectar cambios en los archivos `.ts`.
- **Ejecutar un script npm definido:** `npm run <nombre>` – Ejecuta el comando definido bajo `"scripts"` en package.json. Por ejemplo, `npm run build` ejecutará el script asociado a "build" (usualmente la compilación).
- **Agregar una dependencia en producción:** `npm install <nombre>` – Instala un paquete y lo añade a `dependencies` en package.json. (Para instalarlo en devDependencies, añadir `--save-dev` al comando).

Estos comandos forman parte del flujo de trabajo típico: inicializar el proyecto, instalar librerías (como TypeScript u otras), compilar el código TypeScript a JavaScript, y ejecutar la aplicación o sus pruebas. En conjunto, `package.json` y `tsconfig.json` documentan cómo armar y ejecutar el proyecto, de modo que cualquiera (o cualquier sistema de integración continua) pueda replicar el proceso de construcción sin ambigüedades.

Recursos adicionales

- **Documentación oficial de TypeScript:** Referencia completa y actualizada (en inglés) en el sitio oficial de TypeScript: typescriptlang.org/docs.
- **Tutorial TypeScript de W3Schools:** Introducción práctica a TypeScript (en inglés) en w3schools.com/typescript.