

Material de estudio OBLIGATORIO EJE1 TypeScript

Sitio: [Instituto Superior Politécnico Córdoba](#)
Curso: Programador Web - TSDWAD - 2022
Libro: Material de estudio OBLIGATORIO EJE1 TypeScript

Imprimido por: Braian TRONCOSO
Día: jueves, 4 mayo 2023, 4:45 PM

Descripción



Tabla de contenidos

1. TypeScript

2. Instalación de TypeScript

2.1. ¿Cómo crear y compilar un archivo TypeScript en VSCode?

2.2. ¿Cómo crear un servidor de pruebas en VSCode?

3. ¿Cómo se escribe código en TypeScript?

4. Tipado estático

5. Tipos y subtipos

6. Operadores y Expresiones

7. Estructuras de Decisión (if y switch)

8. Estructuras de Repetición (Loops)

9. Funciones

10. Colecciones Indexadas - Arreglos

11. POO en TypeScript

12. Referencias

1. TypeScript

Introducción

Para saber TypeScript hay que conocer JavaScript.

TypeScript es un lenguaje de programación de código abierto creado por el equipo de Microsoft como una solución al desarrollo de aplicaciones de gran escala con JavaScript dado que este último carece de clases abstractas, interfaces, genéricos, etc. y demás herramientas que permiten los lenguajes de programación tipados. Son ejemplos la compatibilidad con el intellisense, la comprobación de tiempo de compilación, entre otras.

A typescripts se lo conoce además como un superset (superconjunto) de JavaScript ya que es un lenguaje que transpila el fuente de un lenguaje a otro pero, incluye la ventaja de que es verdaderamente orientado a objetos y ofrece además, muchas de las cosas con las que estamos habituados a trabajar los desarrolladores como por ejemplo: interfaces, genéricos, clases abstractas, modificadores, sobrecarga de funciones, decoradores, entre otras varias.

“Los superset compilan en el lenguaje estándar, por lo que el desarrollador programa en aquel lenguaje expandido, pero luego su código es “transpilado” para transformarlo en el lenguaje estándar, capaz de ser entendido en todas las plataformas”(desarrolloweb.com, <https://desarrolloweb.com/articulos/introduccion-a-typescript.html>)

Entonces, si posees algo de conocimiento de JavaScript, ¡tienes ventaja!. Podemos cambiar los archivos de JavaScript a TypeScript de a poco e ir preparando la base del conocimiento para incorporar en su totalidad este nuevo lenguaje.

Diferencias entre TypeScript y JavaScript:

JavaScript	TypeScript
JavaScript se ejecuta en el navegador. Es un lenguaje de programación interpretado.	TypeScript necesita ser “transpilado” a JavaScript, que es el lenguaje entendido por los navegadores.
JavaScript se ejecuta en el navegador, es decir en el lado del cliente únicamente.	TypeScript se ejecuta en ambos extremos. En el servidor y en el navegador.
Es débilmente tipado.	Es fuertemente tipado (tipado estático)
Basado en prototipos.	Orientado a objetos.

Tabla comparativa de los lenguajes JavaScript y TypeScript

2. Instalación de TypeScript

Instalación de TypeScript

Para ejecutar código en JavaScript necesitamos instalar:

- NodeJS, dado que el compilador de Typescript está desarrollado en NodeJS.
- TSC (Command-line TypeScript Compiler), herramienta que permite compilar un archivo TypeScript a JavaScript nativo.

¿Cómo instalar NodeJS?

Para ello, seguir los siguientes pasos:

1. Descargar del sitio oficial <https://nodejs.org/es/> el instalador acorde a su sistema operativo.
2. Instalar **NodeJs**.
 1. Si tienes Windows o Mac, simplemente ejecuta el instalador y ¡listo!
 2. Si tienes Linux, la página de instalación de NodeJS ofrece los comandos para instalar en Linux. Es relativamente sencillo.
3. Evaluar que la instalación fue exitosa. Para ello, ir a la línea de comandos del sistema e introducir el comando: **node**. A continuación, el sistema mostrará por pantalla la versión de NodeJS instalada como sigue:

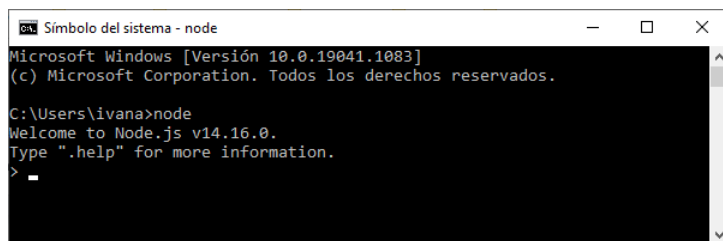


Figura 1: Línea de comandos del sistema Windows después de ejecutar el comando "node".

¿Cómo instalar TSC (Command-line TypeScript Compiler)?

La misma se realizará vía comando **npm** como sigue:

1. Abrir la línea de comandos del sistema.
2. Ejecutar el siguiente comando: **npm install -g typescript**
3. Evaluar que la instalación fue exitosa. Para ello ejecutar el comando: **tsc -v**

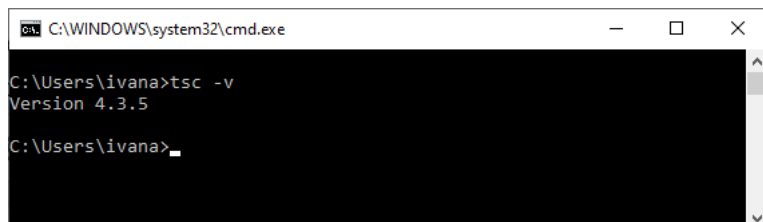


Figura 2: Línea de comandos del sistema luego de ejecutar el comando tsc -v

2.1. ¿Cómo crear y compilar un archivo TypeScript en VSCode?

1. Desde el explorador de archivos, crear un nuevo archivo titulado: **app.ts** dentro de una carpeta app (opcional) como sigue:

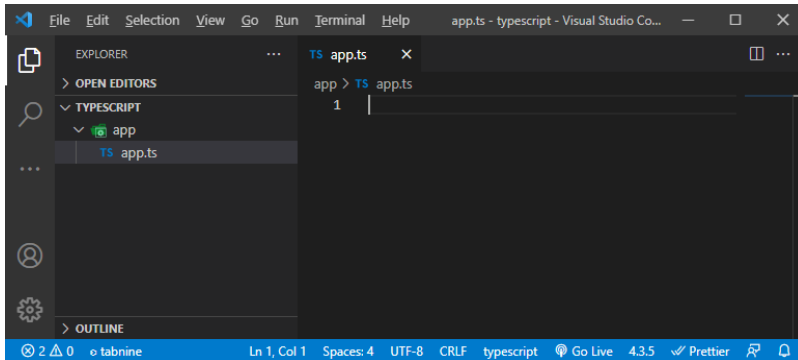


Figura 3: Creación del archivo helloworld.ts

2. Luego, escribir el código TypeScript en dicho archivo:

```
let message: string = 'Hello World';  
  
console.log(message);
```

3. Finalmente, haciendo uso de la terminal de VSCode ejecutar el comando: **tsc app/app.ts**. Este comando compila y si no hay ningún error, crea el nuevo archivo js.

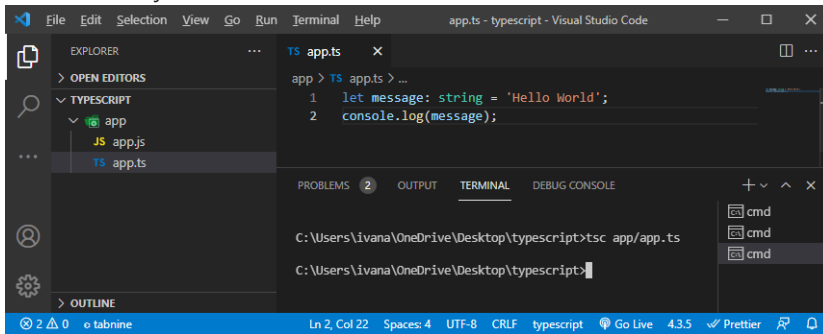


Figura 4: Compilación de TypeScript a JavaScript

Opciones del compilador

“Las opciones del compilador permiten controlar cómo se genera el código JavaScript a partir del código TypeScript de origen. Puedes establecer las opciones en el símbolo del sistema, como haría en el caso de muchas interfaces de la línea de comandos, o en un archivo JSON denominado tsconfig.json.

Hay disponibles numerosas opciones del compilador. Puedes ver la lista de opciones en la lista completa de opciones en la [documentación de las interfaces de la línea de comandos de tsc.](https://docs.microsoft.com/es-es/learn/modules/typescript-get-started/5-typescript-compiler)” (<https://docs.microsoft.com/es-es/learn/modules/typescript-get-started/5-typescript-compiler>)

Para modificar el comportamiento predefinido del TSC con VSCode:

1. Abrir la terminal.
2. Ejecutar el comando: **tsc --init**. A continuación, se creará el archivo **tsconfig.json** como sigue:

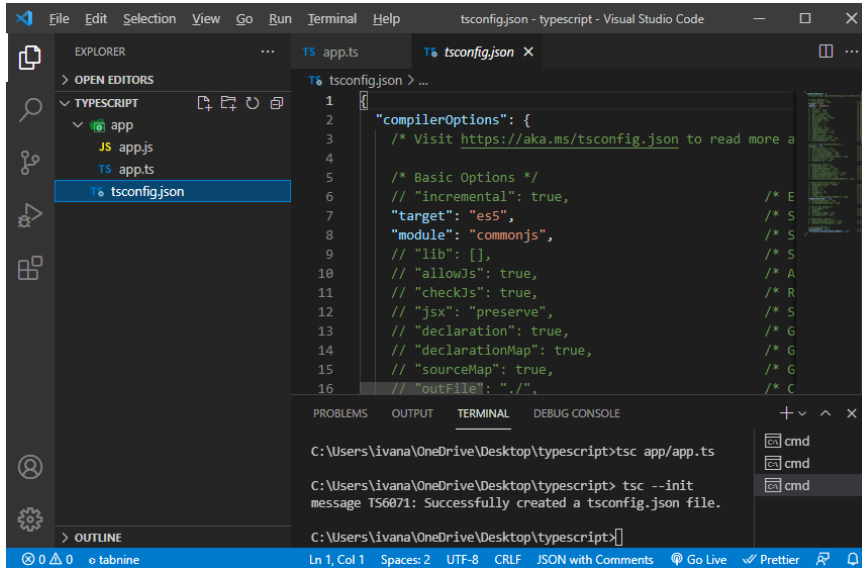


Figura 5: Archivo tsconfig.json generado después de ejecutar el comando tsc --init

3. Editar las configuraciones según se requiera.

Por ejemplo, podemos crear una carpeta que contenga todos los archivos .js generados por el compilador TSC (el output dir/file). Para ello, descomentar la entrada "outFile" y a continuación ejecutar como sigue:

"outFile": "./output/app.js",

y comentamos la entrada "module":

// "module": "commonjs",

y finalmente, ejecutar en la terminal de VSCode el comando **"tsc"**. A continuación, se creará la carpeta **output** conteniendo el archivo **app.js**.

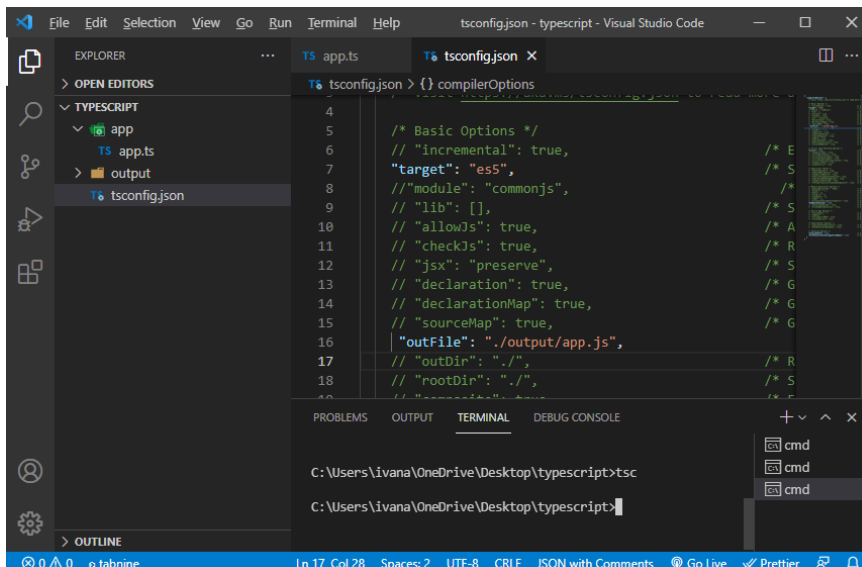
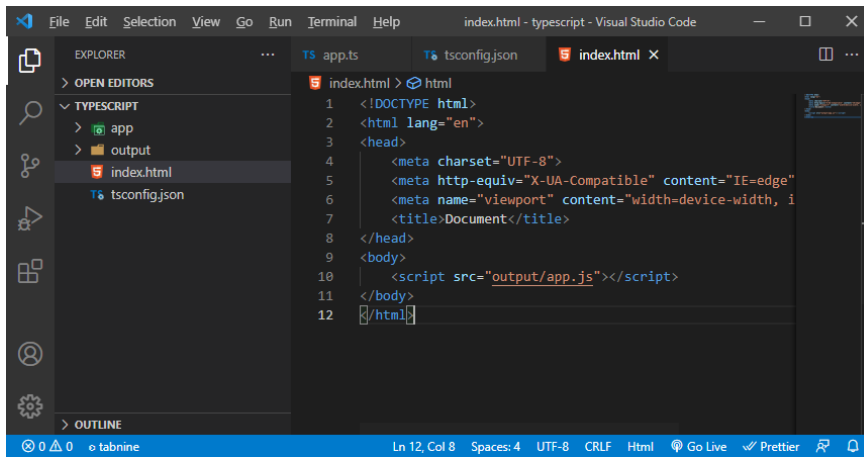


Figura 6: Manipulando la configuración del TSC para que tire los archivos generados a una carpeta output.

Bien, hasta el momento hemos creado un archivo **app.ts** y lo hemos transpilado a un archivo equivalente en JavaScript **app.js** usando el compilador TSC pero, ¿Cómo podemos ejecutarlo para ver los resultados en la consola?

Para ello, realizar los siguientes pasos:

1. Crear un archivo **html** que incluya el script **app.js** como sigue:

The image shows a screenshot of the Visual Studio Code editor. The Explorer sidebar on the left shows a project structure with folders 'app' and 'output', and files 'index.html' and 'tsconfig.json'. The 'index.html' file is selected and its content is displayed in the main editor area. The code is a basic HTML document that includes a script tag pointing to 'output/app.js'.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, 1
7   <title>Document</title>
8 </head>
9 <body>
10   <script src="output/app.js"></script>
11 </body>
12 </html>
```

Figura 7: Archivo index.html

2. Ejecutar el archivo **index.html** o configurar un servidor de prueba para el entorno de desarrollo.

2.2. ¿Cómo crear un servidor de pruebas en VSCode?

Para ello, seguir los siguientes pasos:

1. Ejecutar el siguiente comando **"npm install --global http-server"**. A continuación, se creará la carpeta **node_modules**.
2. Ejecutar el comando **"npm init"**. A continuación, se creará un archivo **package.json**

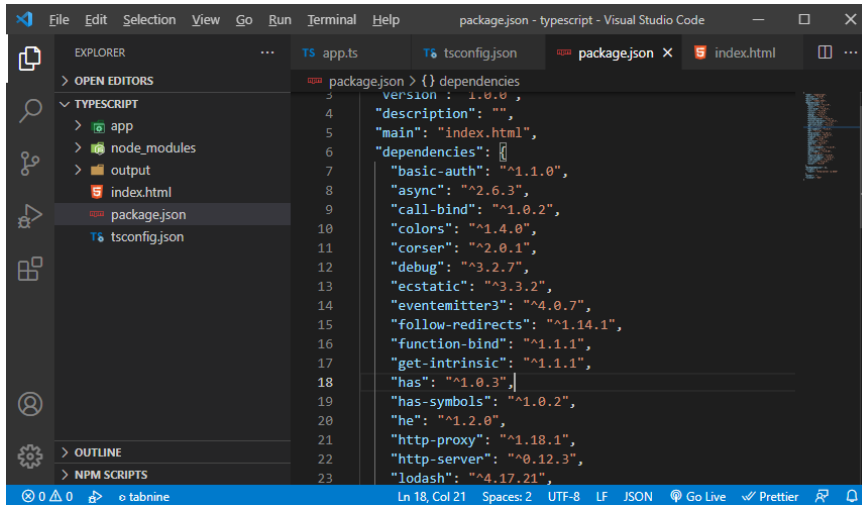


Figura 8: Archivo package.json

Nota: El archivo **package.json** es un archivo que contiene todos los metadatos acerca del proyecto. Son ejemplos: descripción, licencia, autor, dependencias, scripts, entre otros.

3. Configurar la entrada **"scripts"** como sigue:

```
"scripts": {
  "start": "http-server -p 8456"
}
```

4. Finalmente, ejecutar el comando **"npm start"** como sigue:

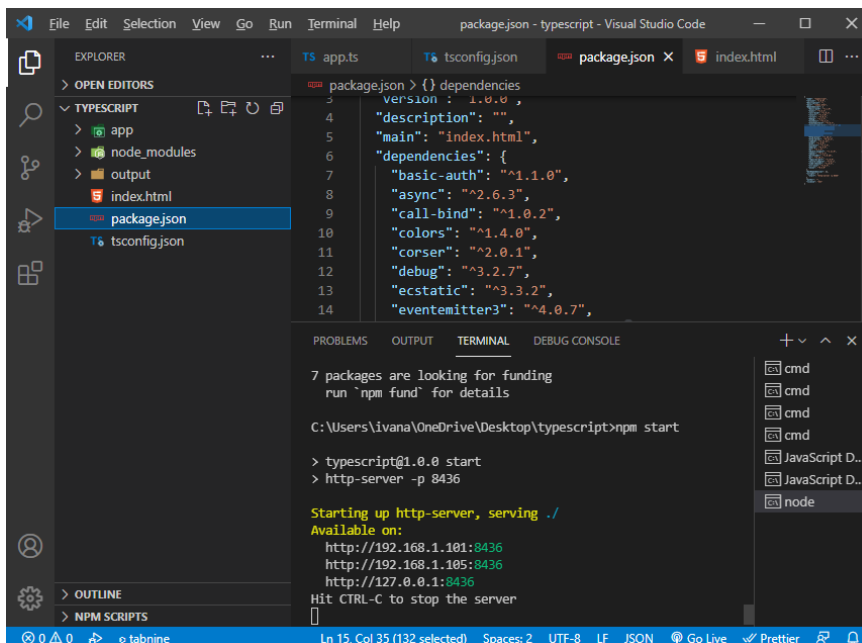


Figura 9: Iniciando el servidor.

Como podemos observar en la Terminal de VSCode, accediendo a la url: <http://127.0.0.1:8436> podremos visualizar nuestro html y, si inspeccionamos el fuente el mensaje "Hola Mundo" en la consola.

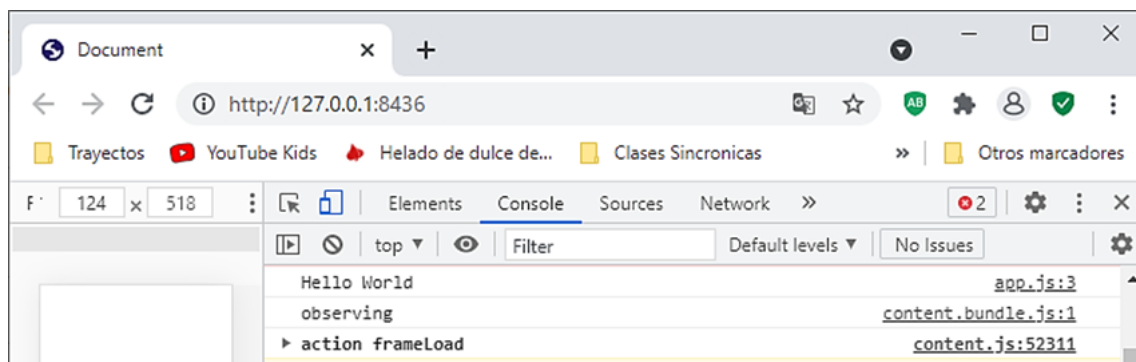


Figura 10: Inspección de código index.html

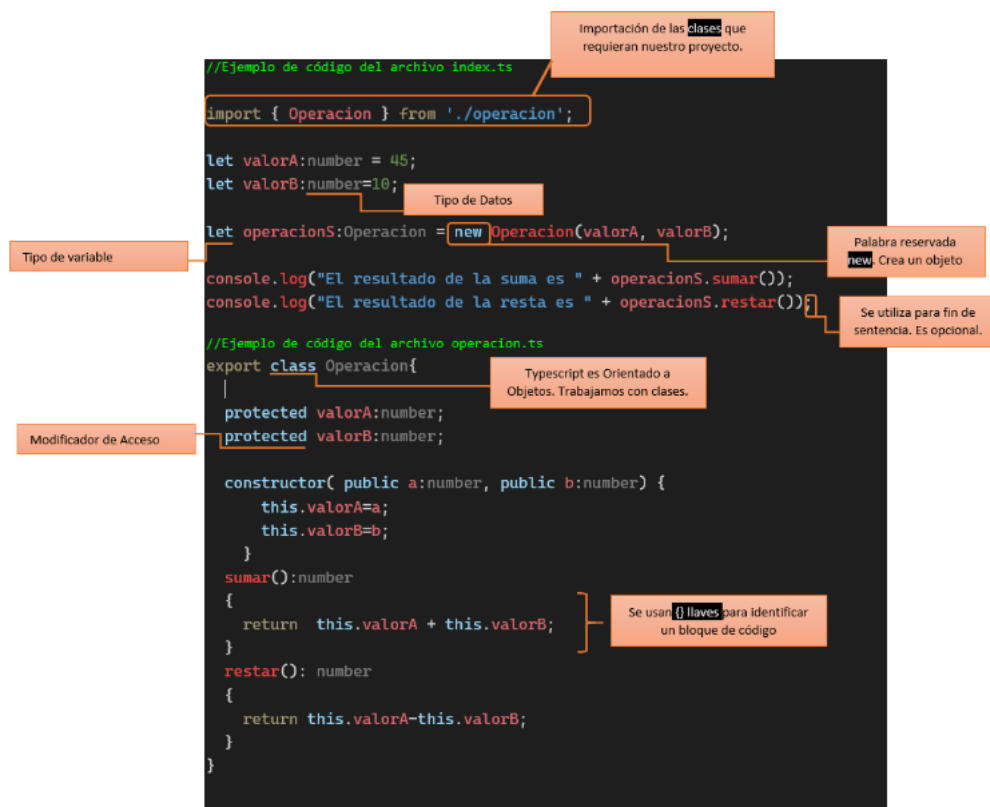
3. ¿Cómo se escribe código en TypeScript?

La sintaxis de un lenguaje de programación define una serie de reglas para escribir código. Cada lenguaje define su propia sintaxis.

En typescript, un programa está compuesto por:

- Módulos
- Funciones
- Clases
- Variables
- Declaraciones y expresiones
- Comentarios
- Entre otros

A continuación, te dejamos una imagen resumen:



Para probar el código fuente click [aquí](#).

Características muy importantes para tener en cuenta al momento de escribir código typescript:

- **Case-sensitive.** Esto significa que Typescript distingue mayúsculas de minúsculas.
- **Nombres de Clases.** Deben estar en PascalCase. Esto significa que los nombres de las palabras que componen el nombre de la clase, deben comenzar en mayúsculas. Ej. MiPrimerClase
- **Nombre de Funciones.** Deben estar en camelCase. Esto significa que los nombres de las palabras que componen el nombre del método o función deben comenzar en mayúsculas a excepción de la primer palabra. Ej. obtenerDatosPersona

Identificadores en Typescript

Los identificadores son los nombres que le damos a los elementos de un programa. Ej. variables, funciones, clases, etc.

Las reglas para los identificadores son:

- Los identificadores pueden contener letras y números. Sin embargo, un identificador no puede comenzar con un número.
- Los identificadores no pueden contener caracteres especiales. A excepción del guión bajo (`_`) y el signo dólar (`$`).
- Los identificadores no pueden ser palabras reservadas del lenguaje.
- Los identificadores deben ser únicos.
- Los identificadores son case-sensitive.
- Los identificadores no pueden contener espacios en blanco.

El siguiente cuadro, muestra algunos ejemplos de identificadores válidos e inválidos.

Identificadores válidos Identificadores inválidos

miNombre	Var
mi_nombre	mi nombre
num1	mi-nombre
\$resultado	1num

Palabras reservadas

break	as	any	switch
case	if	throw	else
var	number	string	get
module	type	instanceof	typeof
public	private	enum	export
finally	for	while	void
null	super	this	new
in	return	true	false
any	extends	static	let
package	implements	interface	function
new	try	yield	const
continue	do	catch	

Espacios en blanco y saltos de línea

Typescript ignora tabs, saltos de línea y líneas en blanco por lo que puedes utilizar tabs, saltos de línea y líneas en blanco para indentar el código en la medida que tú creas que ayuda a la lectura del mismo.

Semicolon (punto y coma) es opcional

El punto y coma que delimita las sentencias es opcional en Typescript. Sin embargo, para una mejor lectura es recomendable utilizarlo.

Comentarios en Typescript

Los comentarios pueden ayudar a comprender el código. ¡Pero recuerda!

“Si tú sientes que es necesario escribir un comentario,

entonces es probable que tengas que refactorizar el código”.

Martin Fowler

Esto significa que no debemos abusar de los comentarios en el código. El código debería explicarse por sí mismo.

Por ende, es recomendable usar los comentarios sólo para incluir información adicional como por ejemplo: el autor del código, sugerencias sobre una función/construcción, etc. El compilador ignora los comentarios.

Typescript soporta dos tipos de comentarios:

Comentarios de una línea. Cualquier texto seguido de `//...`

Comentarios de múltiples líneas. Cualquier texto comprendido entre `/* ... */`

Ejemplo:

```

1  //this is single line comment
2
3  /* This is a
4     |
5     |   Multi-line comment
6     |
7     */

```

4. Tipado estático

Una de las principales características de TypeScript es que es fuertemente tipado por lo que, no sólo permite identificar el tipo de datos de una variable mediante una sugerencia de tipo sino que además permite validar el código mediante la comprobación de tipos estáticos. Por lo tanto, TypeScript permite detectar problemas de código previo a la ejecución cosa que con JavaScript no es posible.

“Los tipos también potencian las ventajas de inteligencia y productividad de las herramientas de desarrollo, como IntelliSense, la navegación basada en símbolos, la opción Ir a definición, la búsqueda de todas las referencias, la finalización de instrucciones y la refactorización del código” (docs.microsoft.com, <https://docs.microsoft.com/es-es/learn/modules/typescript-get-started/2-typescript-overview>)

Se agrega además que TypeScript permite describir mucho mejor el fuente ya que, además de ser tipado, es verdaderamente orientado a objetos (avanzaremos en esto más adelante) lo que permite a los desarrolladores un código más legible y mantenible.

En este punto es importante agregar que si bien TypeScript es muy flexible y puede determinar el tipo de datos implícitamente, es aconsejable utilizar la nomenclatura que recomienda la página oficial (tipado estricto), ya que de este modo permitirá un mejor mantenimiento de nuestro código y el mismo será más legible.

Variables

Antes de avanzar en tipos y subtipos de datos en TypeScript, recordemos el concepto de variable.

Una **variable** es un espacio de memoria que se utiliza para almacenar un valor durante un tiempo (scope) en la ejecución del programa. La misma tiene asociado un tipo de datos y un identificador.

Debido a que TypeScript es un superset de JavaScript, la declaración de las variables se realiza de la misma manera que en JavaScript:

var:

Es el tipo de declaración más común utilizada.

Sintaxis:

```
var medida= 10;
```

```
var m=10;
```

let:

Es un tipo de variable más nuevo (agregado por la [ECMAScript 2015](https://ecma262.github.io/2015/)). El mismo reduce algunos problemas que presentaba la sentencia var en las versiones anteriores de JavaScript.

“Este cambio permite declarar variables con ámbito de nivel de bloque y evita que se declare la misma variable varias veces” (docs.microsoft.com, <https://docs.microsoft.com/es-es/learn/modules/typescript-declare-variable-types/2-types-overview>).

Sintaxis:

```
let precio=0;
```

const:

Es un tipo constante ya que, al asumir un valor no puede modificarse (agregado también por la [ECMAScript 2015](https://ecma262.github.io/2015/))

Sintaxis:

```
const ivaProducto = 0.10;
```

Nota: Como recordatorio, la diferencia entre ellas es que las declaraciones **let** se pueden realizar sin inicialización, mientras que las declaraciones **const** siempre se inicializan con un valor. Y las declaraciones **const**, una vez asignadas, nunca se pueden volver a asignar. (<https://docs.microsoft.com/es-es/learn/modules/typescript-declare-variable-types/2-types-overview>)

Inferencia de tipo en TypeScript

TypeScript permite asociar tipos con variables de manera explícita o implícita como veremos a continuación:

Sintaxis de la inferencia explícita:

```
<variable>: <tipo de datos>
```

Ejemplo inferencia explícita:

```
let edad: number; = 42;
```

Ejemplo inferencia implícita:

```
let edad = 42;
```

Nota: Si bien las asociaciones de tipo explícitas son opcionales en TypeScript, se recomiendan dado que permiten una mejor lectura y mantenimiento del código.

Veamos cómo funciona:

1. Abrir VSCode y crear un nuevo archivo titulado **example01.ts**
2. En dicho archivo, escribir las siguientes declaraciones de variables:

```
let a: number; /* Inferencia explícita
let b: string; /* Inferencia explícita
let c=101;      /* Inferencia implícita
```

En este caso, TypeScript interpreta que la variable **a** es del tipo `number` y **b** del tipo `string` dado que la declaración es explícita. En el caso de la variable **c** infiere que es del tipo `number` dado que éste es el tipo de datos que corresponde al valor con el que se ha inicializado la variable.

Nota: Observa que al posicionar el puntero del mouse sobre la variable **c**, VSCode abre un tooltip con la declaración explícita **"let c:number"**

Pero ¿Qué ocurre si intentamos asignar un tipo de datos diferente a la variable **c**? Para ello, escribir a continuación la siguiente línea:
c="one";

VSCode marca un error en la línea de la asignación y en el explorador puedes ver el archivo en rojo. Observa además que al posicionar el puntero del mouse sobre la línea VSCode muestra el mensaje de error **"Type string is not assignable to type number"**.

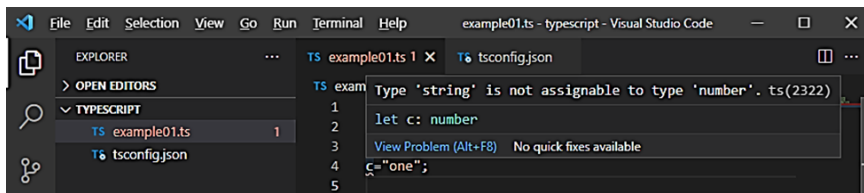


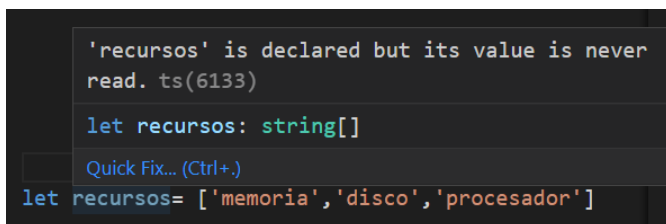
Figura 12: Error TypeScript. Asignación de tipos.

Analicemos otro ejemplo:

Dada la siguiente declaración:

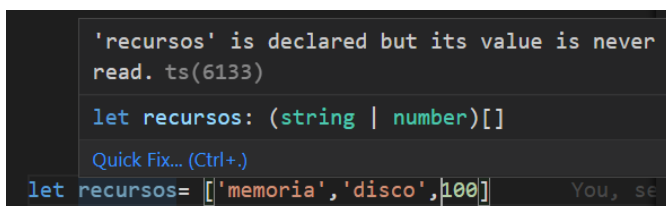
```
let recursos: ['memoria', 'disco', 'procesador'];
```

Hasta aquí sabemos que: "let", es la declaración de un arreglo cuyo nombre es "recursos" y el tipo no ha sido definido explícitamente.



Entonces, si posicionamos el puntero del mouse sobre la variable "recursos" podemos observar que TypeScript automáticamente entiende por sí solo que se trata de un arreglo del tipo `String`.

Sin embargo, si luego introducimos en el array un valor de otro tipo, y posicionamos el puntero del mouse sobre la variable **recursos**, podremos observar que el arreglo admite variables del tipo "string" o (símbolo para "o" es "|") "number".



Y quizás, este no sea el comportamiento que deseamos por ello, se aconseja como buena práctica especificar el tipo de datos de manera explícita.

En este caso:

```
let recursos: string [] = ['memoria', 'disco', 'procesador']
```

De esta manera, si intentamos introducir un elemento `number` u otro tipo a nuestro arreglo, el compilador nos marcará un error.

Ámbito de la Variable

El ámbito de las variables depende del lugar dónde la declaramos como puedes observar en la siguiente imagen:

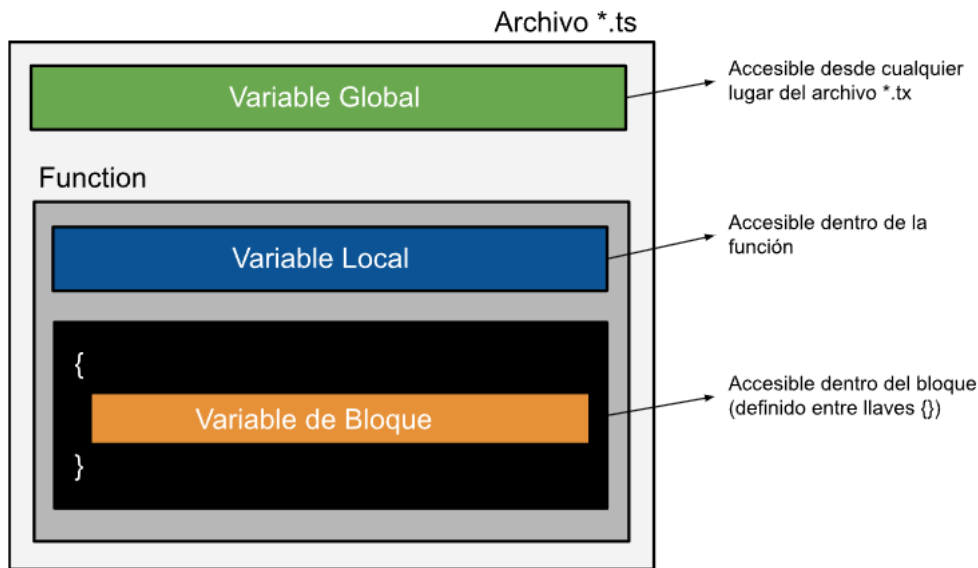


Figura 13: Ambito de variables en typescript.

Como puedes observar, cuando declaramos una variable fuera de la función se denomina **variable global** dado que ésta, estará disponible para cualquier otro código en el documento actual. Por otro lado, cuando declaramos una variable dentro de una función se denomina **variable local**, porque estará disponible sólo dentro de esa función donde fue creada.

Antes de ECMAScript 6 no consideraba el ámbito de sentencias de bloque; más bien, una variable declarada dentro de un bloque es local para la función (o ámbito global) en la que reside el bloque. Esto podría traer inconvenientes en algunos casos. Por eso, el estándar ECMAScript 2015 introdujo `let`.

A partir de la versión ES6 / ECMAScript 2015 habilita tres ámbitos para la declaración de una variable:

Ámbito Global: Cuando se declara una variable fuera de una función, se le denomina variable global, porque está disponible para cualquier otro código en el documento actual.

Ámbito Función: Cuando se declara una variable dentro de una función, se le denomina variable local, porque está disponible sólo dentro de esa función donde fue declarada.

Ámbito Bloque: Este nuevo ámbito es utilizado para las variables declaradas con la palabra reservada `let` / `const` dentro de un bloque de código delimitados por llaves `{ }`, esto implica que no pueden ser accesibles fuera de este bloque.

Ejemplo de Ambios de Variable:

```
let global_num = 12          //variable global
class Numbers {
  almacenarNumero():void {
    var local_num = 14;      //variable local
  }
}
console.log("Número Ámbito Global: " + global_num )
var obj = new Numbers();
}
```

5. Tipos y subtipos

Todos los tipos en TypeScript son subtipos de un único tipo principal denominado tipo `any`. `Any` es un tipo que representa cualquier valor de JavaScript sin restricciones.

Any:

Puede ser de cualquier tipo y su uso está justificado cuando no tenemos información a priori de qué tipo de dato se trata. Este tipo de definición es propia de TypeScript.

Sintaxis:

```
let cantidad: any = 4;
let desc: any [] =[1,true,"verde"]
```

Todos los demás tipos se clasifican como primitivos, de objeto o parámetros. Estos tipos presentan diversas restricciones estáticas en sus valores.

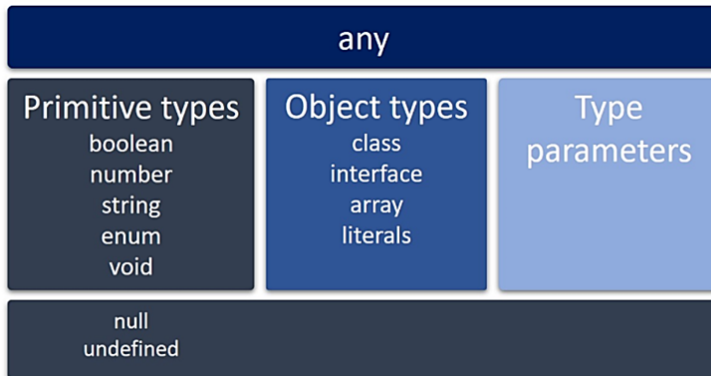


Figura 14: Tipos y subtipos

(Fuente de la imagen, <https://docs.microsoft.com/en-us/learn/modules/typescript-declare-variable-types/2-types-overview>)

Tipos Primitivos

Los tipos primitivos son: boolean, number, string, void, null, undefined y enum.

string:

Representa valores de cadena de caracteres (letras);

Sintaxis:

```
let saludo: string = "hola, mundo";
```

TypeScript permite también usar plantillas de cadenas con las que podemos intercalar texto con otras variables: `${ expr }`

Ejemplo:

```
let nombre: string = "Mateo";
let mensaje: string = `Mi nombre es ${nombre}.
Soy nuevo en TypeScript.`;
console.log(mensaje);
```

number:

Representa valores numéricos, como enteros (int) o decimales (float).

Sintaxis:

```
let codigoProducto: number = 6;
```

boolean:

Es un tipo de variable que puede tener solo dos valores, Verdadero (true) o Falso (false).

Sintaxis:

```
let estadoProducto: boolean = false;
```

Void:

El tipo `void` existe únicamente para indicar la ausencia de un valor, como por ejemplo en una función que no devuelve ningún valor.

Sintaxis:

```
function mensajeUsuario(): void {

    console.log("Este es un mensaje para el usuario");

}
```

Enum:

“Las enumeraciones ofrecen una manera sencilla de trabajar con conjuntos de constantes relacionadas. Un elemento enum es un nombre simbólico para un conjunto de valores. Las enumeraciones se tratan como tipos de datos y se pueden usar a fin de crear conjuntos de constantes para su uso con variables y propiedades.

Siempre que un procedimiento acepte un conjunto limitado de variables, considere la posibilidad de usar una enumeración. Las enumeraciones hacen que el código sea más claro y legible, especialmente cuando se usan nombres significativos” (docs.microsoft.com, <https://docs.microsoft.com/es-es/learn/modules/typescript-declare-variable-types/4-enums>).

Ejemplo:

```
/**Crear la enumeración */
enum Color {
    Blanco,
    Rojo,
    Verde
}

/**Declarar la variable y asignar un valor de la enumeración */
let colorAuto: Color = Color.Blanco;

console.log(colorAuto); //return 0
```

Para observar el fuente, lo que imprime la consola y el fuente JavaScript generado, hacer clic [aquí](#).

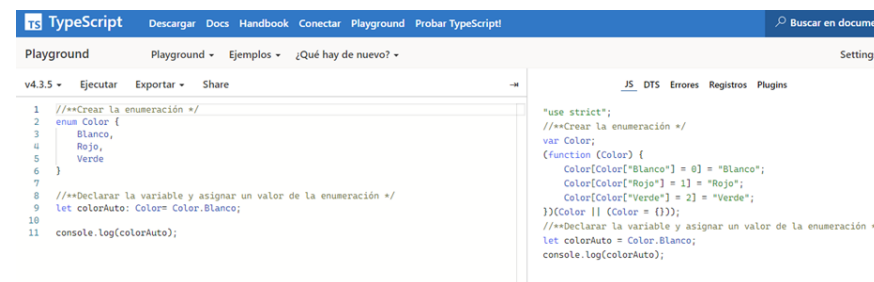


Figura 15: Ejemplo del tipo enum en TypeScript y su transpilación a JavaScript.

<https://www.typescriptlang.org/play> nos provee una herramienta para ver la transpilación a JavaScript e incluso ejecutarla. Para ello, simplemente hacer clic en Ejecutar.

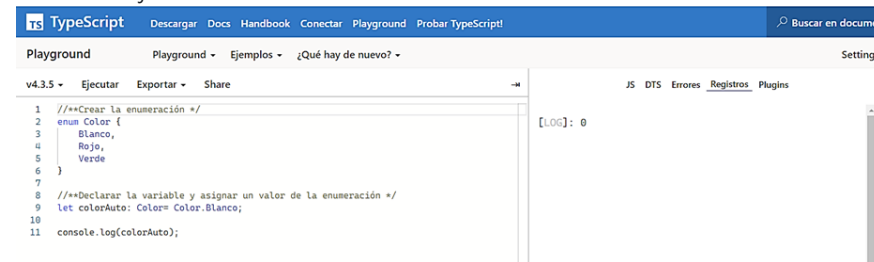


Figura 16: Ejecución de TypeScript con la herramienta <https://www.typescriptlang.org/play>

Como puedes observar en la figura 8 y 9, tras ejecutar nuestro enum en TypeScript y transpirarlo a JavaScript, este se transforma el tipo enum (TypeScript) a una función (JavaScript) y devuelve un nro. que va desde 0 en adelante, de acuerdo a la opción elegida.

Tipos de objetos

Los tipos de objeto son todos los tipos de clase, de interfaz, de arreglos y literales.

Nota: Los tipos de clase e interfaz se abordarán más adelante en este mismo módulo.

Array:

Es un tipo de colección o grupos de datos (vectores, matrices). El agrupamiento lleva como antecesor el tipo de datos que contendrá el arreglo.

Ejemplo:

```
number[]

String[]
```

Sintaxis:

```
let list : number[]=[1,2,3];
```

Ejemplos:

```
let list : string[]=['pimiento','papas','tomate'];
let rocosos: boolean[] = [true, false, false, true]
```

```
let perdidos: any[] = [9, true, 'asteroides'];
```

```
let diametro: [string, number] = ['Saturno', 116460];
```

Generic:

También puedes definir tipos genéricos como sigue:

```
function identity<T>(arg: T): T {
    return arg;
}
```

Los genéricos son como una especie de plantillas mediante los cuales podemos aplicar un tipo de datos determinado a varios puntos de nuestro código. Sirven para aprovechar código, sin tener que duplicarlo por causa de cambios de tipo y evitando la necesidad de usar el tipo "any" (<https://desarrolloweb.com/articulos/generics-typescript.html>).

Los mismos se indican entre "mayores y menores" y pueden ser de cualquier tipo incluso clases e interfaces.

Veamos el ejemplo que nos provee la fuente oficial de TypeScript: <https://www.typescriptlang.org/docs/handbook/generics.html>:

Si tenemos la siguiente función:

```
function identity(arg: number): number {
    return arg;
}
```

Pero, necesitamos que la misma sea válida para otros tipos de datos entonces podríamos cambiar el tipo number por any como sigue:

```
function identity(arg: any): any {
    return arg;
}
```

Sin embargo, el tipo any permite cualquier tipo de valor por lo que la función podría recibir un tipo number y devolver otro. Entonces, estamos perdiendo información sobre el tipo que debe devolver la función. Para solucionarlo, y obligar al compilador que respete el mismo tipo (parámetros de entrada y salida) podemos utilizar genéricos.

```
function identity<T>(arg: T): T {
    return arg;
}
```

Observa que cambiamos any por la letra **T**.

T nos permite capturar el tipo de datos por lo que el tipo utilizado para el argumento es el mismo que el tipo de retorno.

Object:

Es un tipo de dato que engloba a la mayoría de los tipos no primitivos.

Sintaxis:

```
let persona:object={nombre:"Ana", edad:45}
```

Desestructuración

La desestructuración permite acceder a los valores de un array o un objeto.

Ejemplo - desestructuración de un objeto:

```
var obj={a:1,b:2,c:3};
console.log(obj.c);
```

Ejemplo - desestructuración de un array:

```
var array=[1,2,3];
console.log(array[2]);
```

Ejemplo - desestructuración con estructuración:

```
var array=[1,2,3,5];
var [x,y, ...rest]= array;
console.log(rest);
```

Observa que la sintaxis **...rest**, nos permite agregar más parámetros. En este caso el resultado en consola será: [3, 5]

Puedes comprobar el fuente [aquí](#) (clic en Ejecutar para observar lo que muestra en pantalla console.log)

Estructuración

Como se pudo observar en el apartado anterior, la estructuración facilita que una variable del tipo array reciba una gran cantidad de parámetros.

Ejemplo en funciones:

```
function rest(first, second, ...allOthers)
{console.log(allOthers);}
```

Observa que la sintaxis **...allOthers** nos permite pasar más parámetros.

Luego, al llamar a la función con los siguientes parámetros:

```
rest('1', '2','3','4','5'); //return 3,4,5
```

Tipos null y undefined

“Los tipos null y undefined son subtipos de todos los demás tipos. No es posible hacer referencia explícita a los tipos null y undefined. Solo se puede hacer referencia a los valores de esos tipos mediante los literales null y undefined” (docs.microsoft.com, <https://docs.microsoft.com/es-es/learn/modules/typescript-declare-variable-types/2-types-overview>).

Aserción de tipos (As)

Una aserción de tipos le indica al compilador “**confía en mí, sé lo que estoy haciendo**”. Se parece al casting en otros lenguajes de programación pero no tiene impacto en tiempo de ejecución sino que le dice al compilador el tipo de datos en cuestión a fin de acceder a los métodos, propiedades, etc. del tipo de datos en tiempo de desarrollo.

Sintaxis (dos posibles)

```
(nombre as string).toUpperCase();

(<string>nombre).toUpperCase();
```

6. Operadores y Expresiones

Un operador define alguna función que se realizará con los datos. Los datos sobre los que trabajan los operadores se denominan operandos.

Considera la siguiente expresión:

3 + 5 = 8

Aquí los símbolos 3, 5 y 8 son **operandos** mientras que los símbolos + e = son **operadores**.

TypeScript clasifica los operadores como sigue:

- Aritméticos
- De Comparación (o relacionales)
- Lógicos
- De Asignación
- Condicional Ternario

Operadores Aritméticos

(Asume que los valores de las variables a y b son 10 y 5 respectivamente.)

Operador	Descripción	Ejemplo
+ (Adición)	Realiza la operación de suma y retorna la suma de los operandos.	a + b dará 15
- (Resta)	Realiza la operación de resta y retorna la diferencia de los operandos.	a - b dará 5
* (Multiplicación)	Realiza la operación de multiplicación y retorna el producto de los operandos	a * b dará 50
/ (División)	Realiza la operación de división y retorna el cociente	a / b dará 2
% (Módulo)	Realiza la operación de división y retorna el resto	a % b dará 0
++ (Incremento)	Incrementa el valor del operando en 1 unidad.	a++ dará 11
-- (Decremento)	Decrementa el valor del operando en 1 unidad.	a-- dará 9

Puedes ver un ejemplo haciendo click [aquí](#).

Operadores de Comparación (o relacionales)

Los operadores relacionales comparan dos entidades y devuelven un valor booleano, es decir, verdadero/falso.

Asume que el valor de A es 10 y B es 20.

Operador	Descripción	Ejemplo
== Igualdad	Devuelve verdadero (true) si ambos operandos son iguales.	(A==B) es falso
!= Desigualdad	Devuelve verdadero (true) si ambos operandos no son iguales.	(A != B) es verdadero.
>= Mayor que	Devuelve verdadero (true) si el operando de la izquierda es mayor o igual que el operando de la derecha.	(A >= B) es falso
> Mayor	Devuelve verdadero (true) si el operando de la izquierda es mayor que el operando de la derecha.	(A > B) es falso

<	Menor	Devuelve verdadero (true) si el operando de la izquierda es menor que el operando de la derecha.	(A < B) es verdadero
= <	Menor que	Devuelve verdadero (true) si el operando de la izquierda es menor o igual que el operando de la derecha	(A <= B) es verdadero

Puedes ver un ejemplo haciendo click [aquí](#).

Operadores Lógicos

Los operadores lógicos se utilizan para combinar dos o más condiciones y retornan también un valor booleano: true o false.

Asume que A es 10 y B es 20.

Operador	Descripción	Ejemplo
&& (lógico Y)	El operador Y (denominado AND) retorna true sólo si todas las condiciones son true	(A > 10 && B > 10) es falso
(lógico O)	El operador O (denominado OR) retorna true si al menos una de las condiciones es true	(A > 10 B > 10) es verdadero
! (lógico no)	El operador no (denominado NOT) retorna el valor inverso de la expresión !(A > 10) es verdadero resultante.	

Puedes ver un ejemplo haciendo click [aquí](#).

Operadores de Asignación

El operador de asignación es un operador binario que se utiliza para asignar el valor de su operando de la derecha a una variable, propiedad o elemento de indexador que proporciona el operando de la izquierda.

Operator	Description	Example
= (Asignación Simple)	Asigna valores del operando del lado derecho al operando del lado izquierdo	C = A + B asignará el valor de A + B a C
+= (Agrega y Asigna)	Suma el operando derecho al operando izquierdo y asigna el resultado al operando izquierdo.	C += A es equivalente a C = C + A
-= (Resta y Asigna)	Resta el operando derecho del operando izquierdo y asigna el resultado al operando izquierdo.	C -= A es equivalente a C = C - A
*= (Multiplica y Asigna)	Multiplica el operando derecho con el operando izquierdo y asigna el resultado al operando izquierdo.	C *= A es equivalente a C = C * A
/= (Divide y Asigna)	Divide el operando izquierdo con el operando derecho y asigna el resultado al operando izquierdo.	C /= A es equivalente a C = C / A

Puedes ver un ejemplo haciendo click [aquí](#).

Operador condicional (ternario)

El operador condicional es el único operador de Typescript que necesita tres operandos. El operador asigna uno de dos valores basado en una condición. La sintaxis de este operador es:

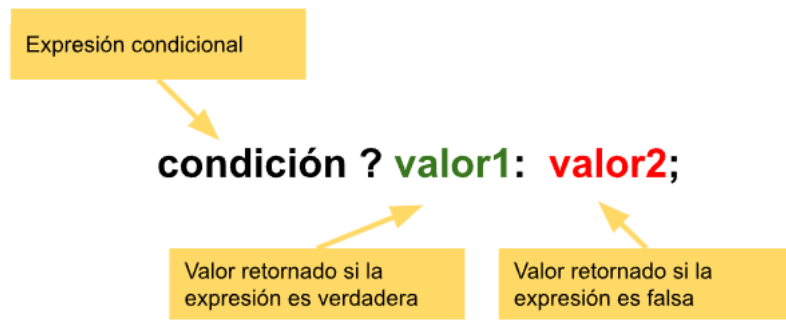


Figura 17: Operador Condicional

Si la condición es true, el operador tomará el valor1, de lo contrario tomará el valor2. Puedes usar el operador condicional en cualquier lugar que use un operador estándar.

Por ejemplo:

```
let edad:number=5;
let estado:string = (edad>=18) ? "Adulto" : "Menor";
console.log(estado);
```

El resultado: "Menor"

7. Estructuras de Decisión (if y switch)

Las estructuras de decisión requieren que el programador especifique una o más condiciones, junto con un bloque de sentencias que se ejecutarán si la condición es verdadera y, opcionalmente otro bloque de sentencias si la condición es falsa.

A continuación, podemos observar un diagrama de flujo con su correspondiente código asociado:

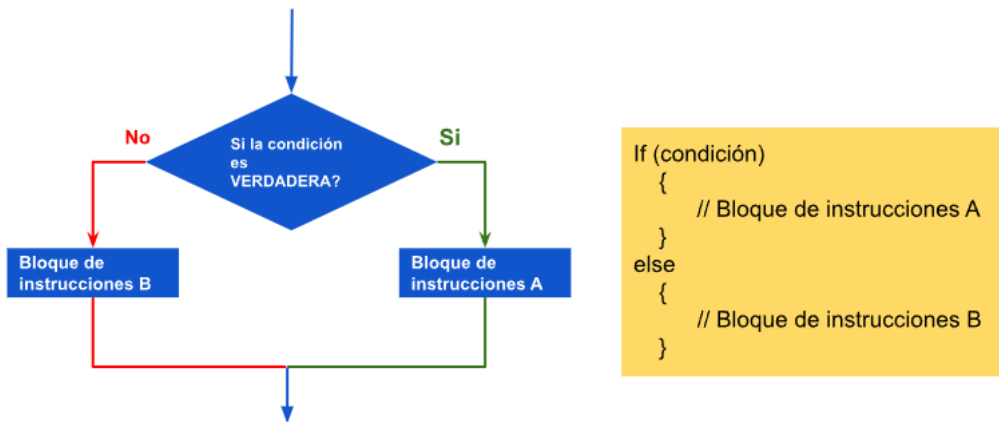
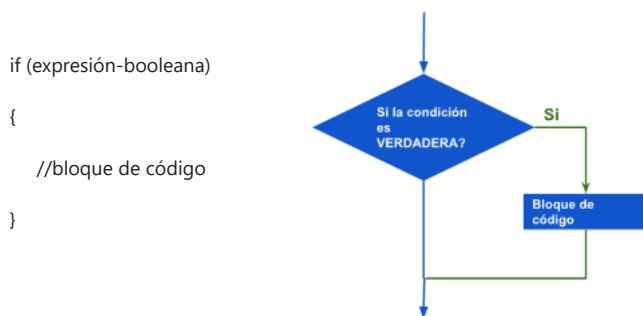


Figura 18: Estructura Condicional

En Typescript podemos trabajar la estructura de control condicional o (de toma de decisiones) de la siguiente manera:

Declaración if



Si la expresión booleana se evalúa como verdadera, entonces se ejecutará el bloque de código dentro de la estructura de control if.

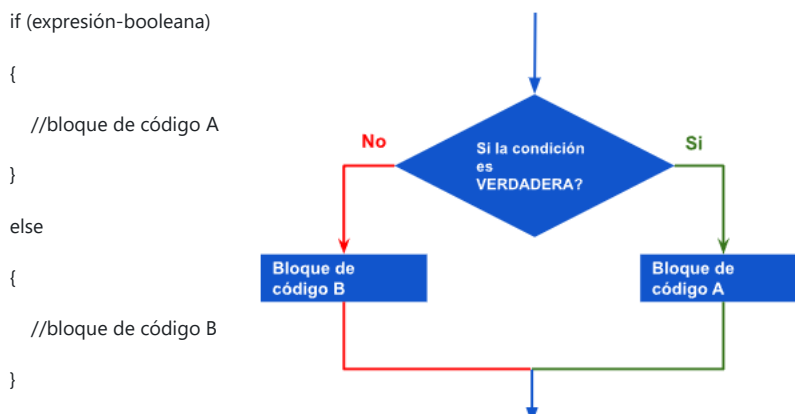
Ejemplo:

```

1 let num:number = 5
2 if (num > 0) {
3     console.log("number is positive")
4 }
5

```

Declaración if else



Si la expresión booleana se evalúa como verdadera, entonces se ejecutará el bloque de código A. Si la expresión booleana se evalúa como falsa, se ejecutará el primer conjunto de código B.

Ejemplo:

```
1 let num = 12;
2 if (num % 2 == 0) {
3   console.log("Par");
4 } else {
5   console.log("Impar");
6 }
```

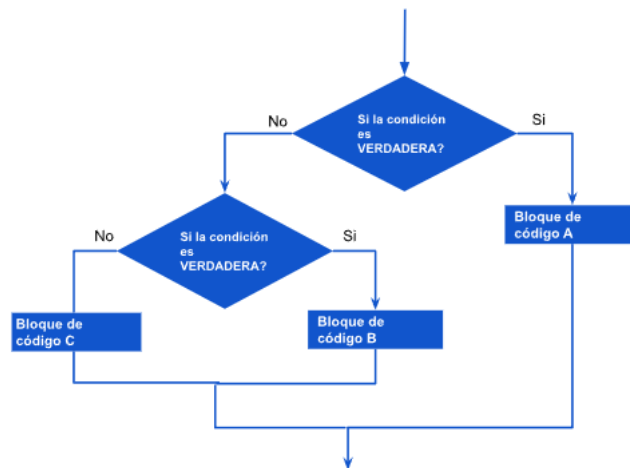
Declaración else if

if (expresión-booleana)

```
{
  //bloque de código A
}

else if (expresión-booleana)
{
  //bloque de código B
}

else
{
  //bloque de código C
}
```



Si deseas evaluar o probar múltiples condiciones, es posible usar una declaración 'if' o 'else if' dentro de otra declaración 'if' o 'else if'.

Consideraciones a tener en cuenta en este caso:

- Un 'if' puede tener cero o más 'else' y éste, debe venir después de un 'if'.
- Un 'if' puede tener cero o más 'else if' y éste, debe venir después antes del 'else'.
- Una vez que el 'else if' sucede, ninguno de los otros 'else if' o 'else' se evaluará o probará.

Ejemplo:

```
1 let num:number = 2
2 if(num > 0) {
3   console.log(num + " es positivo")
4 } else if(num < 0) {
5   console.log(num + " es negativo")
6 } else {
7   console.log(num + " es nulo")
8 }
```

Declaración switch


```

switch(variable-expresión) {

  case constant_expr1: {

    //bloque de código A;

    break;

  }

  case constant_expr2: {

    //bloque de código B;

    break;

  }

  default: {

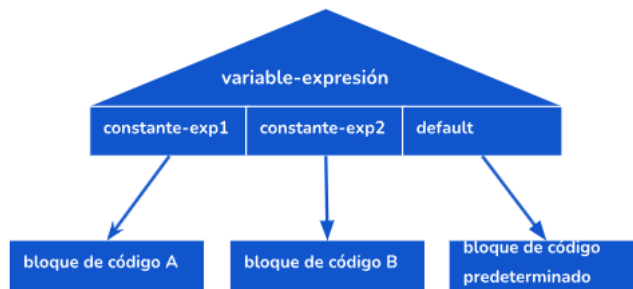
    //bloque de código Predeterminado;

    break;

  }

}

```



La declaración 'switch' permite que la variable se pruebe contra una lista de valores. El valor de variable-expresión se prueba en todos los casos. Si la misma coincide con uno de los casos, se ejecuta el bloque de código correspondiente (A o B). Si ninguna coincide con el valor de variable-expresión, se asocia al bloque de código predeterminado.

Ejemplo:

```

1  let nota:string = "A";
2  switch(nota) {
3      case "A": {
4          console.log("Excelente");
5          break;
6      }
7      case "B": {
8          console.log("Bien");
9          break;
10     }
11     case "C": {
12         console.log("Regular");
13         break;
14     }
15     case "D": {
16         console.log("Insuficiente");
17         break;
18     }
19     default: {
20         console.log("Invalida opción");
21         break;
22     }
23 }

```

¡Pongámonos a prueba!

Modifica el siguiente código (que identifica el mayor de dos números) a fin de encontrar ahora el mayor de 3 números.

```
1 let num1:number=12;
2 let num2:number=-5;
3
4 if (num1 > num2)
5 {
6 console.log("El mayor de los dos números es: " + num1);
7 }
8 else
9 {
10 console.log("El mayor de los dos números es: " + num2)
11 }
```

Puedes trabajar en stackblitz: <https://stackblitz.com/edit/typescript-bvnnc3?file=index.ts>

Es posible que el trabajo no haya sido del todo fácil, sobre todo si trabajaste anidado if.

Desafío ¿Existe otra forma de resolver el mayor de los tres números? ¿Cuál será la mejor forma y por qué?

Si aún no lo descubres, intenta con operadores lógicos...

¡Importante! Como puedes observar, en este simple ejemplo, nosotros como profesionales del área de desarrollo, siempre nos vamos a encontrar con más de una forma de resolver y en muchas ocasiones, algunas formas de resolver serán más acertadas que otras... entonces ¿Cómo identifico cuál es la mejor forma de resolver?

A continuación te dejamos algunos puntos a tener en cuenta para tomar una decisión:

- La probabilidad de fallos. El código simple y/o autogenerado siempre es menos propenso a fallos.
- La performance. Una aplicación web lenta va directo al fracaso.
- La escalabilidad. Debe ser simple y fácil de mantener. Debe ser código limpio.
- El tiempo de desarrollo. El tiempo que nos va a llevar es un factor también a medir dado que el tiempo, para la empresa es dinero.
- Entre otras.

Valores falsos

Los siguientes valores se evalúan como falso (también conocidos como valores Falsy):

- false
- undefined
- null
- 0
- NaN
- la cadena vacía ("")

Ejemplo:

```
1 function comprobarId() {
2     if (document.getElementById("id")?.innerHTML != "") {
3         return true;
4     } else {
5         alert("idHidden está vacío");
6         return false;
7     }
8 }
```

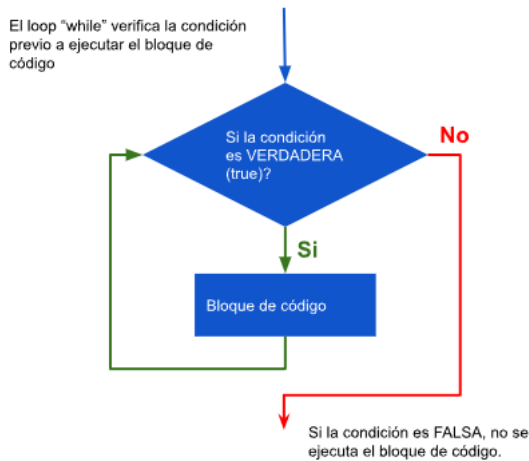
Como puedes observar, la función comprobarId devuelve true si el elemento id tiene asociado un objeto Text; en caso contrario, muestra una alerta y devuelve false.

8. Estructuras de Repetición (Loops)

Muchas veces, nos encontramos en situaciones dónde necesitamos que un bloque de código se ejecute varias veces. Una estructura de control repetitiva (bucle o loop) nos permite ejecutar un bloque de código varias veces.

TypeScript, como todos los lenguajes de programación, provee varias estructuras de control repetitivas (bucles o loops) que permiten ejecutar un bloque de código n veces.

A continuación, la forma más general de estructura de control repetitiva (bucle o loop) denominada Mientras (o while loop).



```
while(expresión-booleana) {
  // bloque de código.
}
```

Figura 19: Estructura de Repetición

Como puedes observar, el bloque de código se ejecutará MIENTRAS la condición sea VERDADERA.

TypeScript provee diferentes tipos de loops para manejar las repeticiones de bloques de código. La siguiente imagen lo ilustra:

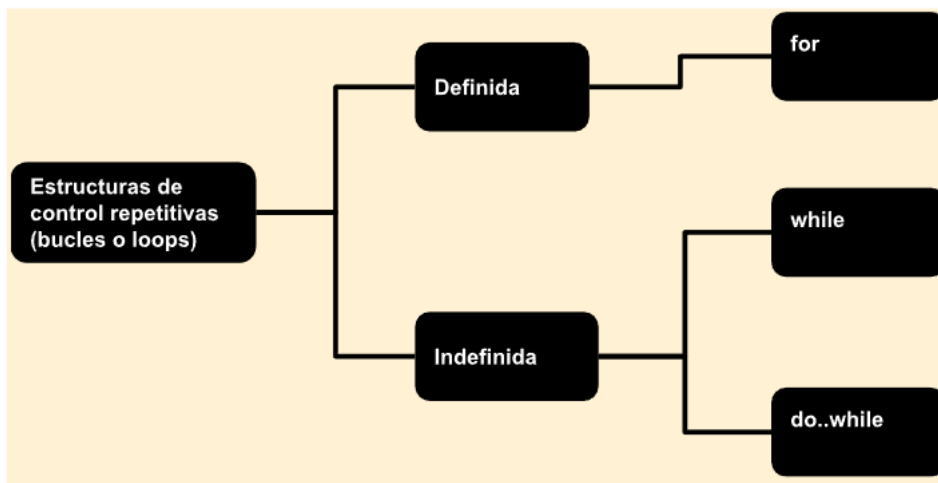


Figura 20: Tipos de Loops

Estructuras de Control Definidas

Se llama estructura de control definidas a un loop cuyo número de iteraciones está predefinido (se sabe de antemano).

For loop

La estructura de control PARA o for loop ejecuta un bloque de código un número específico de veces. Esta estructura de control puede ser utilizada para iterar una lista o matriz.

Sintaxis:

for (inicialización; condición; incremento)

```
{
  //bloque de código
}
```

Cuando for loop se ejecuta, ocurren los siguientes pasos:

1. La expresión de inicialización, si existe, se ejecuta. Esta expresión habitualmente inicializa uno o más contadores del bucle, pero la sintaxis permite una expresión con cualquier grado de complejidad. Esta expresión puede también declarar variables.
2. Se evalúa la expresión condición. Si el valor de condición es true, se ejecuta la sentencia del bucle. Si el valor de condición es false, el bucle for finaliza. Si la expresión condición es omitida, la condición es asumida como verdadera.

Se ejecuta la sentencia. Para ejecutar múltiples sentencias, usar las llaves ({ ... }) para agruparlas.

Se ejecuta la expresión incremento, si hay una, y el control vuelve al paso 2.

Ejemplo:

```
1 for(let i:number =0; i< 10;i++) {
2   console.log(i);
3 }
```

Desafío ¿Es posible declarar una estructura de control for sin la expresión de inicialización y sin expresión de incremento? ¿Qué consideraciones debería tener en cuenta para trabajar la estructura de control for de esta forma?

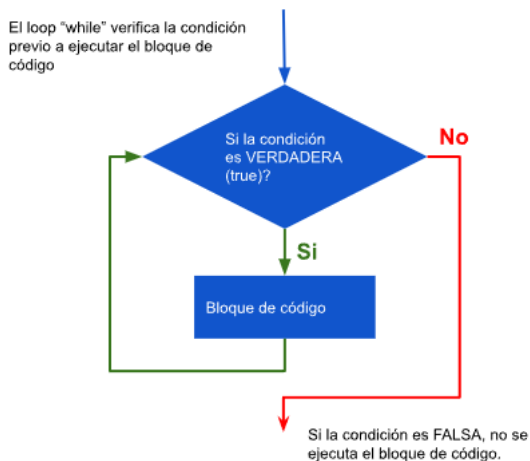
Ejercitación

Haciendo uso de la estructura de control for intenta:

1. Calcular el factorial de un número.
2. Calcular la sumatoria de los números comprendidos en el rango [10,50]
3. Calcular el promedio de un número.

While loop

La estructura de control MIENTRAS (o while loop) ejecuta un conjunto de sentencias (bloque de código) MIENTRAS la condición sea evaluada como verdadera.



```
while(expresión-booleana) {
  // bloque de código.
}
```

Figura 21: While

Sintaxis:

```
while(condición) {

  // bloque de código

}
```

Es importante agregar que, la condición se evalúa antes de que la sentencia contenida en el bucle sea ejecutada por lo tanto, si la condición es evaluada como falsa, se detiene la ejecución y el control pasa a la sentencia siguiente al while.

Ejemplo:

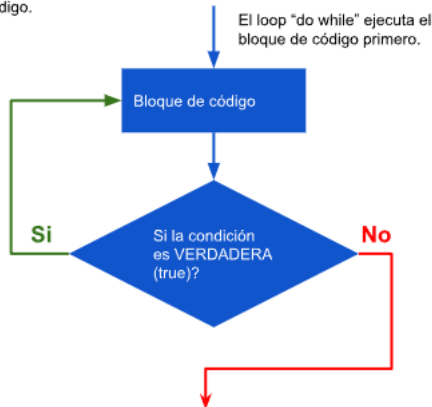
```
1 let num:number = 5;
2 let factorial:number = 1;
3
4 while(num >=1) {
5   factorial = factorial * num;
6   num--;
7 }
8 console.log("El factorial es: "+factorial);
9
```

Nota: Evita los bucles infinitos. Asegúrate de que la condición en el while loop llegue finalmente a ser falsa; de otra forma, el loop nunca terminará. Las sentencias en el siguiente loop while se ejecutan sin fin, porque la condición nunca llega a ser falsa.

Do while loop

La estructura de control HASTA (o do while loop) ejecuta un conjunto de sentencias (bloque de código) HASTA que la condición sea evaluada como falsa.

El loop "do while" verifica la condición luego de ejecutar el bloque de código.



```
do {
    // Bloque de código
} while(expresión-booleana);
```

Figura 22: Do While

Sintaxis:

```
do {
    //bloque de código
} while(condición)
```

Es importante agregar que, la condición se evalúa después de que la sentencia contenida en el bucle sea ejecutada por lo tanto, si la condición es evaluada como verdadero repite la ejecución de las sentencias dentro del bloque de código o bien si es falsa, se detiene la ejecución y el control pasa a la sentencia siguiente al do while.

Ejemplo:

```
1 let n:number = 10;
2 do {
3     console.log(n);
4     n--;
5 } while(n>=0);
```

Ejemplo: While vs Do While loop

```
1 let n:number = 5
2 while(n > 5) {
3     console.log("Se ejecutó el bloque de sentencias del MIENTRAS");
4 }
5 do {
6     console.log("Se ejecutó el bloque de sentencias del HASTA");
7 }
8 while(n>5)
9
```

El ejemplo declara inicialmente un ciclo while. El bloque de código sólo se ejecuta si la condición $n > 5$ se evalúa como verdadera. En el presente ejemplo, el valor de n no es mayor que cero, por lo que la expresión devuelve falso y saca el control del bucle. Es decir, no se ejecuta la instrucción `console.log("Se ejecutó el bloque de sentencias del MIENTRAS");`

Por otro lado, el loop `do...while` ejecuta la declaración una vez. Esto se debe a que la iteración inicial no considera la expresión booleana. Sin embargo, para la iteración posterior, while comprueba la condición y saca el control del bucle.

La sentencia label

La sentencia label proporciona un identificador que permite referirse a un punto en el código o sentencia desde cualquier lugar de su programa. Por ejemplo, podemos usar un label para identificar un loop, y usar las sentencias `break` o `continue` para indicar si el programa debe interrumpir un loop o continuar su ejecución.

Sintaxis:

label:

sentencia de código

Ejemplo:

```
2 markLoop:
3 while (theMark == true) {
4     doSomething();
5 }
```

En el ejemplo se puede observar que el label markLoop identifica a un bucle while.

La sentencia break

La sentencia break para salir de un loop, switch, o en conjunto con una sentencia label.

Cuando usamos la sentencia break sin un label, la instrucción finaliza inmediatamente el código encerrado en while, do-while, for, o switch y transfiere el control a la siguiente sentencia.

Por otro lado, cuando usamos la sentencia break con un label, la instrucción termina la sentencia especificada por label.

Sintaxis:

break;

break label;

Ejemplo:

El siguiente ejemplo itera hasta que encuentra el primer múltiplo de 5, en ese momento finaliza el while (es decir, sale del loop):

```
1 let i:number = 1
2 while(i<=10) {
3     if (i % 5 == 0) {
4         console.log ("El primer múltiplo de 5 entre 1 y 10 es: "+i)
5         break //sale del loop
6     }
7     i++
8 }
```

La sentencia continue

A diferencia de la sentencia break, la sentencia continue no sale del loop sino que da curso a la iteración siguiente.

Sintaxis:

continue;

Ejemplo:

```
1 let num:number = 0
2 let count:number = 0;
3
4 for(num=0;num<=30;num++) {
5     if (num % 2==0) {
6         continue
7     }
8     count++
9 }
10
11 console.log ("La cantidad de valores pares entre 0 y 30 es: "+ count);
```

El ejemplo anterior muestra el número de valores impares comprendidos entre 0 y 30.

Desafío. La variable count es una variable especial denominada en el mundo de la programación "contador". ¿Por qué se llamará así?

También existen otras variables especiales: acumulador y bandera. Investiga qué significa y cuál es su utilidad. Finalmente, comparte con la clase o por medio de los foros un ejemplo dónde se explique y demuestre su utilidad.

9. Funciones

Una función es un **conjunto de instrucciones** o sentencias que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente y se caracterizan porque:

- deben ser invocadas por su nombre.
- permiten **simplificar el código** haciendo más legible y reutilizable.

La declaración de una función consiste en:

- Un nombre
- Una lista de parámetros o argumentos encerrados entre paréntesis.
- Conjunto de sentencias o instrucciones encerradas entre llaves.

Sintaxis:

```
function nombre (parámetro1, parámetro2)
{
  /**instrucciones a ejecutar */
}
```

Ejemplo:

```
1 function potenciaDeDos(num:number) {
2   return num* num;
3 }
```

La función potenciaDeDos toma un argumento, llamado num y retorna el argumento de la función multiplicado por sí mismo. La sentencia return especifica el valor que retorna la función.

Los parámetros primitivos (como puede ser un número) son pasados a las funciones por valor; el valor. Esto significa que, si la función cambia el valor del parámetro, este cambio no es reflejado globalmente o en otra llamada a la función.

Por el contrario, si se pasa un objeto (p. ej. un valor no primitivo, como un Array o un objeto definido por el usuario) como parámetro (por referencia), y la función cambia las propiedades del objeto, este cambio sí es visible desde afuera de la función.

Llamar a una función

Definir una función no la ejecuta. Definir una función simplemente la nombra y especifica qué hacer cuando la función es llamada. Llamar la función es lo que realmente realiza las acciones especificadas con los parámetros indicados. Por ejemplo, si define la función square, podría llamarla como sigue:

potenciaDeDos(3);

Retornar datos

Como mencionamos más arriba, las funciones pueden devolver valores junto con el control a quién lo llama. Estas funciones suelen llamarse funciones de retorno.

Sintaxis:

```
function nombre_funcion():tipo_de_datos_de_retorno {

  //bloque de código

  return value;
}
```

dónde:

- tipoDeRetorno, puede ser cualquier tipo de datos.
- return value, una función debe retornar al menos un valor.
- el tipo de datos debe emparejarse con el tipoDeRetorno especificado en la función.

Ejemplo:

```
1 function saludo():string { //la función retorna un string
2   return "Hola Mundo" ;
3 }
```

Parámetros

A partir de ECMAScript 6, hay dos nuevos tipos de parámetros: Parámetros por defecto y los parámetros REST.

Parámetros por defecto

En typescript, los parámetros de funciones están establecidos por defecto: undefined. Sin embargo, en ciertas situaciones puede ser útil establecerlos a un valor suministrado por defecto diferente. Es entonces cuando los parámetros por defecto pueden ayudar.

Sintaxis:

```
function nombre_funcion(parametro1=valor_por_defecto, parametro2, ...) {  
  
    //bloque de código  
  
}
```

Parámetros Rest

La sintaxis de parámetros rest (resto) nos permite representar un número indefinido de argumentos en forma de array. Es decir que, no nos limitan la cantidad de parámetros que se le puede pasar a una función. Sin embargo, los parámetros deben ser todos del mismo tipo.

Sintaxis:

Para declarar un parámetro rest, en nombre del parámetro debe tener un prefijo de tres puntos (...). Cualquier parámetro que no sea rest debe ir antes.

```
function nombre_funcion(parametro1, ...parametrosRest:tipo_de_datos[])  
  
{  
  
    //bloque de código  
  
}
```

Ejemplo:

```
1  function sumatoria(...nums:number[]) {  
2      let i:number;  
3      let sum:number = 0;  
4  
5      for(i = 0;i<nums.length;i++) {  
6          sum = sum + nums[i];  
7      }  
8      console.log("sum of the numbers",sum)  
9  }  
10 sumatoria(1,2,3);  
11 sumatoria(1,10,10,5,11);
```

Observa que la función sumatoria acepta n números y, calcula la sumatoria de los mismos.

Parámetros opcionales

Tal como su nombre lo indica, los parámetros opcionales se suelen utilizar cuando no es necesario pasar argumentos obligatoriamente.

Un parámetro se puede marcar como opcional agregando el signo de interrogación a su nombre. Es importante mencionar que, el parámetro opcional se debe establecer como último argumento de la función.

Sintaxis:

```
function funcion_nombre(param1:tipo_de_datos, param2?:tipo_de_datos);
```

Ejemplo:

```
1  function obtenerDatosPersonales(id:number,nombre:string,mail?:string) {  
2      console.log("ID:", id);  
3      console.log("Name",nombre);  
4      if(mail!=undefined)  
5          console.log("Email Id",mail);  
6  }  
7  obtenerDatosPersonales(123,"Cami");  
8  obtenerDatosPersonales(456,"Mai","mai@mail.com");
```

El ejemplo mostrado arriba, declara una función parametrizada. Como puedes observar, el tercer parámetro es opcional por lo que no pasarlo, no implica un error en tiempo de compilación y, el valor de la variable mail será undefined consecuentemente.

Funciones Anónimas

Typescript permite las llamadas funciones anónimas. Es decir, que no tienen un nombre. Estas funciones se declaran dinámicamente en tiempo de ejecución. Es decir, que podemos asignar a una variable una función anónima.

Las funciones anónimas, tal como las funciones estándar, pueden aceptar parámetros y devolver resultados.

Sintaxis:

```
let variable_resultado = function( [arguments] ) { ... }
```

Ejemplo:

```
let suma= function(a: number, b: number) { return a + b; }
```

```
console.log(suma(2, 3)); // Salida: 5
```

Desafío!. Investiga ¿cuándo sería conveniente utilizar funciones anónimas?

10. Colecciones Indexadas - Arreglos

El uso de variables para almacenar valores posee limitaciones dado que, las variables tal como las conocemos hasta ahora, son de naturaleza escalar. Es decir que, la declaración de una variable solo puede almacenar un dato a la vez. Por ende, para almacenar n valores en un programa se necesitarán arreglos. Un arreglo no es más que una colección homogénea de valores. ¿Qué quiere decir esto? Que es una colección de valores del mismo tipo de datos.

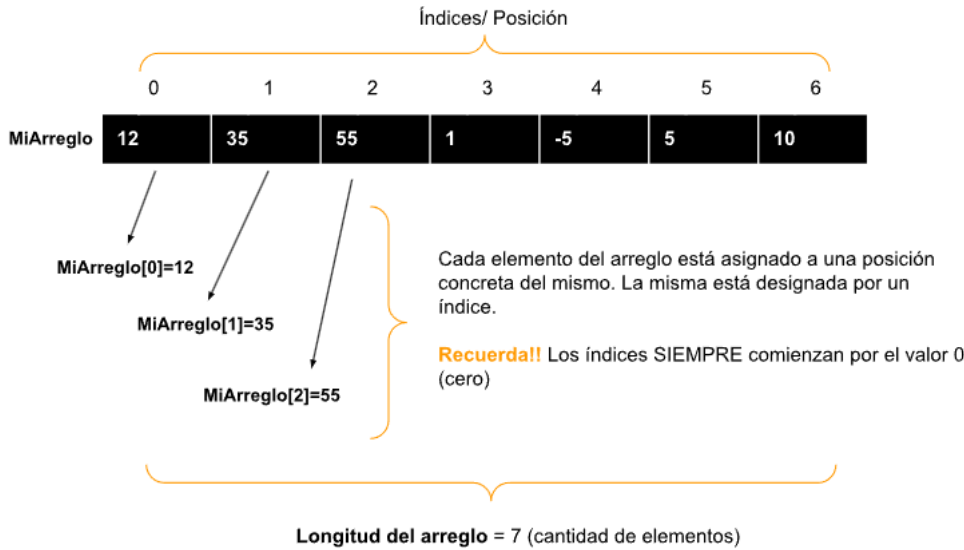


Figura 23: Arreglos

Características

- Las declaraciones de un arreglo, asignan bloques de memoria secuenciales.
- Los arreglos son estáticos. Es decir que, una vez inicializada, no se puede cambiar de tamaño. Es decir que, los arreglos son estructuras fijas. ¿Qué quiere decir esto? Simplemente, si inicializamos el arreglo con n índices o posiciones, estas se mantendrán sin posibilidad de cambiar su tamaño.
- Cada bloque de memoria, representa un elemento del arreglo.
- Los elementos del arreglo se identifican mediante un número entero único llamado índice del elemento.
- Al igual que las variables, los arreglos también deben declararse antes de usarse. Usar la palabra clave var o let para declarar el arreglo.
- Los valores de los elementos del arreglo se pueden actualizar o modificar, pero no se pueden eliminar.

Tipos de Arreglos en TypeScript

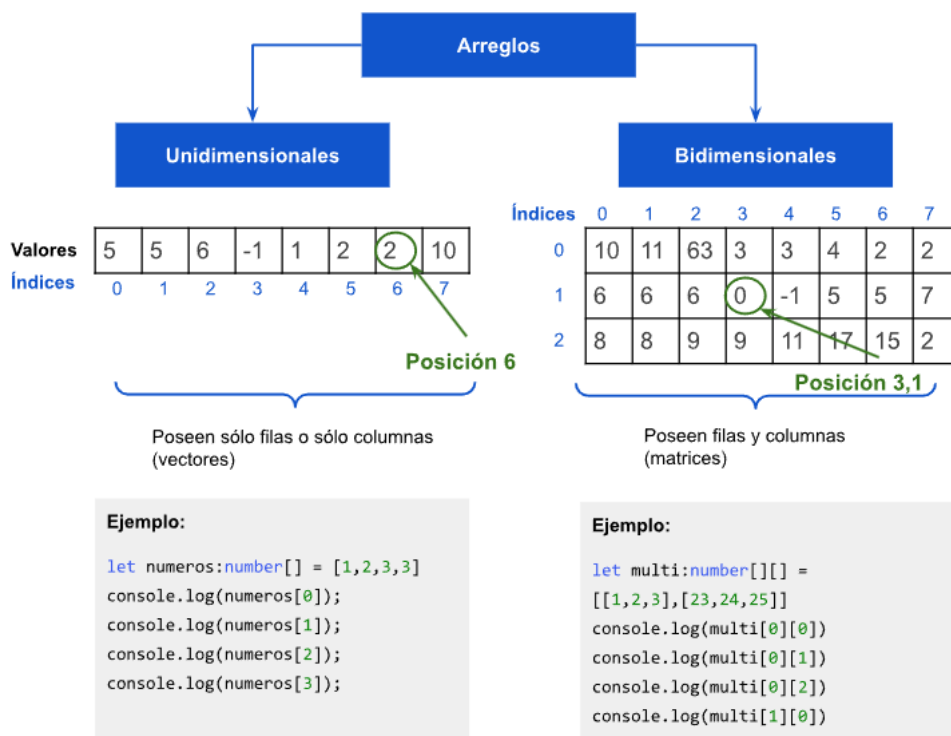


Figura 24: Tipos de Arreglos

Declaración e inicialización de arreglos

TypeScript nos provee dos formas de definir arreglos.

La primera es indicando el tipo de los elementos del arreglo y, añadiendo el operador [] como sigue:

Sintaxis:

let nombre_arreglo:tipo_de_datos[]; //declaración

nombre_arreglo= [dato1,dato2,dato3..] //inicialización

let nombre_arreglo:tipo_de_datos[] = [val1,val2...valN] //declaración e inicialización en la misma línea.

Ejemplo:

```
1 let numeros:number[];
2 numeros= [1,3,4,5] ;
3 console.log(numeros[0]);
4 console.log(numeros[1]);
```

Una declaración de arreglo sin especificar el tipo de datos se considera del tipo any. El tipo se deducirá del primer elemento del arreglo durante la inicialización.

Por ejemplo, una declaración como – var numlist:[] = [2,4,6,8] ,typescript creará un arreglo de números.

Nota: el par de [] especifica la dimensión del arreglo.

Y la segunda, a través de la clase Array.

Sintaxis:

let nombre_arreglo:tipo_de_datos[]= new Array(longitud_del_arreglo) //declaración

nombre_arreglo= [dato1,dato2,dato3..]; //inicialización

let nombre_arreglo:tipo_de_datos[]=new Array(valor1, valor2.. valorN); //declaración e inicialización.

Ejemplo 1:

```
1 let numPares:number[] = new Array(10)
2 for(var i = 1;i<numPares.length;i++) {
3     numPares[i] = i * 2
4     console.log(numPares[i])
5 }
```

Ejemplo 2:

```
8 let nombres:string[] = new Array("Ana","Maira","Camila","Andrés")
9 for(let i = 0;i<nombres.length;i++) {
10     console.log(nombres[i])
11 }
```

Acceder a un elemento del arreglo

El nombre del arreglo seguido de su índice permite acceder al valor del elemento.

Sintaxis:

nombreArreglo[indice] = value

Ejemplo:

```
1 let numList:string[];
2 numList= ["1","2","3","4"]
3 console.log(numList[0]);
4 console.log(numList[1]);
```

Figura 23: Arreglos

Método y Descripción	Ejemplos
----------------------	----------

concat()

Retorna un nuevo arreglo con los nuevos valores que le pasamos como parámetros a la función.

```
let nombres:string[] = new Array("Rosario", "Juan");
console.log( nombres.concat("Ana", "José", "Maria"));
```

every()

Retorna true o false si todos los elementos pasan la prueba definida en la función.

```
let resultado = [12, 5, 8, 130, 44].every(esPar);
console.log("Resultado de la Prueba: " + resultado );
function esPar(elemento:number, index:number, array:any) {
    return (elemento % 2 == 0);
}
```

filter()

Retorna un nuevo arreglo con todos los elementos para los que la función de filtrado devuelve verdadero.

```
function esPar(elemento:number, index:number, array:any) {
    return (elemento %2 == 0);
}
```

```
let resultado = [12, 5, 1, 130, 44].filter(esPar);
```

```
console.log("Sólo Pares : " + resultado );
```

```
let num= [1, 2, 3];
```

forEach()

Llama a una función por cada elemento del arreglo.

```
num.forEach(function (value) {
    console.log(value);
});
```

indexOf()

Retorna el primer índice de un elemento dentro del arreglo igual al valor especificado, o -1 si no se encuentra ninguno.

```
let indice= [3, 5, 8, -8, 50];
console.log("El indice es : " + indice.indexOf(8));
```

join()

Une todos los elementos del arreglo en una cadena.

```
let miArreglo = new Array("Uno","Dos","Tres");
let miCadena = miArreglo.join();
```

```
console.log("str : " + miCadena );
```

Nota: Por defecto el separador que utilizará Typescript es coma (.). Si deseas utilizar otro simplemente puedes pasarlo por parámetro. Ej. miArreglo.join("*")

lastIndexOf()

Retorna el último índice de un elemento dentro de la matriz igual al valor especificado, o -1 si no se encuentra ninguno.

```
let indice = [12, 5, 8, 130, 8].lastIndexOf(8);
console.log("El indice es: " + indice );
```

map()

Crea un nuevo arreglo con los resultados de llamar a una función proporcionada en cada elemento de este arreglo.

```
let numeros = [1, 4, 9];
let raices = numeros.map(Math.sqrt);
console.log("La raíz de los números son: " + raices );
```

pop()

Elimina el último elemento del arreglo y devuelve ese mismo elemento.

```
let numeros = [1, 4, 9];
let elemento = numeros.pop();
console.log("El elemento eliminado del arreglo es: " + elemento );
```

push()	let numeros = new Array(1, 4, 9);
Agrega uno o más elementos al final del arreglo y devuelve la nueva longitud del arreglo.	let longitud = numeros.push(10); console.log("Los números en el arreglo son: " + numeros); console.log("La nueva longitud del arreglo es: " + longitud);
reduce()	
Aplica la función simultáneamente contra dos valores del arreglo (de izquierda a derecha) para reducirla a un solo valor.	let total = [0, 1, 2, 3].reduce(function(a, b){ return a + b; }); console.log("el total es: " + total);
reduceRight()	
Aplice la función simultáneamente contra dos valores del arreglo (de derecha a izquierda) para reducirla a un solo valor.	let total = [0, 1, 2, 3].reduceRight(function(a, b){ return a + b; }); console.log("total is : " + total);
reverse()	let miArreglo= [0, 1, 2, 3].reverse();
Invierte el orden de los elementos del arreglo.	console.log("Arreglo invertido: " + miArreglo);
shift()	
Elimina el primer elemento del arreglo y devuelve ese elemento.	let miArreglo = [10, 1, 2, 3].shift(); console.log("El elemento eliminado es: " + miArreglo);
slice()	
Extrae una sección de un arreglo y devuelve un nuevo arreglo.	let colores = ["verde", "azul", "blanco", "celeste", "rojo"]; console.log("colores.slice(1, 3) : " + colores.slice(1, 3));
	function esMayorQueDiez(elemento:any, indice:number, miArreglo:number[]) { return (elemento >= 10); }
some()	
Devuelve true si al menos un elemento del arreglo satisface la función de prueba proporcionada.	let pruebaUno = [2, 5, 8, 1, 4].some(esMayorQueDiez); console.log("El valor retornado es: " + pruebaUno); let pruebaDos = [12, 5, 8, 1, 4].some(esMayorQueDiez); console.log("El valor retornado es: " + pruebaDos);
sort()	let colores = ["verde", "azul", "blanco", "celeste", "rojo"];
Ordena los elementos del arreglo.	console.log(colores.sort());
splice()	let colores = ["verde", "azul", "blanco", "celeste", "rojo"];
Agrega y/o elimina elementos de un arreglo.	console.log(colores.splice(1,0,"rosa")); console.log(colores);

toString()

Devuelve una cadena que representa el arreglo y sus elementos.

Devuelve una cadena que representa la matriz y sus elementos.

unshift()

Agrega uno o más elementos al inicio de un arreglo y devuelve la nueva longitud del arreglo.

```
let colores = ["verde", "azul", "blanco", "celeste", "rojo"];
```

```
var longitud = colores.unshift("fuxia");
```

```
console.log("Nuevo Arreglo: " + colores);
```

```
console.log("La longitud del arreglo es: " + longitud);
```

Tuplas

En ocasiones, es necesario almacenar colecciones de valores de distintos tipos. Para ello, typescript nos brinda un tipo de datos llamado tuplas.

Una tupla representa una colección heterogénea de valores. Es decir que, nos permiten almacenar múltiples valores de diferente tipo de datos.

Nota: Las tuplas también se pueden pasar como parámetros a funciones.

Sintaxis:

```
let miTupla=[tipoDato1, tipoDato2, tipoDato3, ...tipDatoN];
```

Ejemplo:

```
let miTupla= [28, "Rosario", "López"];
```

Ejemplo:

```
1 let tupla: [string, number, string];
2 tupla = ['hola', 3, 'adios'];
3 console.log(tupla[0]); // salida: hola
4 console.log(tupla[1]); // salida: 3
5 console.log(tupla[2]); // salida: adios
```

Operaciones con Tuplas

Método y Descripción	Ejemplos
push()	Ejemplo:
Agrega un elemento a la tupla.	let miTupla = [28, "Rosario", "López"];
	miTupla.push(2022);
pop()	
Remueve y retorna el último valor de la tupla.	

push()

Ejemplo:

```
let miTupla = [28, "Rosario", "López"];
```

Agrega un elemento a la tupla.

```
miTupla.push(2022);
```

pop()

Remueve y retorna el último valor de la tupla.

Modificar elementos de una tupla

Las tuplas son mutables. Es decir que, se pueden actualizar los valores.

Ejemplo:

```
1 let miTupla: [number, string, string]= [10,"Hello","World"];
2 console.log("Valor de la tupla en el index 0: "+miTupla[0])
3 //actualizando el valor miTupla[0] = 2022
4 console.log("Valor de la tupla en el index 0 modificado a: "+miTupla[0])
```

11. POO en TypeScript

A continuación abordaremos el tema de **programación orientada a objetos** ya que son de suma importancia para trabajar con **TypeScript** y consecuentemente en **Angular**.

¿Qué es la Programación orientada a objetos?

La Programación Orientada a Objetos (POO) es un paradigma de programación que utiliza objetos como elementos fundamentales en la construcción de un producto de software permitiendo que, el código sea más claro y manejable lo que facilita el desarrollo de aplicaciones complejas.

¿Qué es una clase?

En pocas palabras una clase no es más que una plantilla o modelo que define las características y comportamientos que tendrán los objetos que se creen a partir de ella.

Sintaxis:

```
class MiClase
```

```
{
}
```

Ejemplo:

Modificadores
de Acceso

```
class Persona{
    readonly nombre:string;
    readonly apellido:string;
    private añoNac:number;
    constructor(nombre:string, apellido:string) {
        this.nombre = nombre;
        this.apellido = apellido;
    }
    public toString():string
    {
        return this.nombre + this.apellido;
    }
    public edad(añoActual:number):number
    {
        return ( añoActual - this.añoNac);
    }
}
```

Atributos

Constructor

Métodos

Figura 25: Clase Persona

- **Atributos:** Son variables que se declaran dentro de la clase, y sirven para indicar la forma o características de cada objeto representado por esa clase. Los atributos, de alguna manera, muestran lo que cada objeto es, o también, lo que cada objeto tiene.

Sintaxis:

```
<nombre_variable>: <tipo_de_datos>
```

- **Métodos:** Son funciones, procedimientos o rutinas declaradas dentro de la clase, usados para describir el comportamiento o las acciones de los objetos descriptos por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto hace.

Sintaxis:

```
<nombre_método>(<parámetros>): <tipo_de_datos_devuelto>
```

```
{
    /**instrucciones*/
}
```

- **Constructores:** es un método especial que permite instanciar un objeto. Su nombre está definido por la palabra constructor, y no tiene ningún tipo de retorno. Puede recibir 0 a n parámetros.

Sintaxis:

```
Constructor( (<parámetros>: <tipo_de_datos_devuelto>))
```



```
{
  /**instrucciones*/
}
```

- **Propiedades (getters y los setters).** Las mismas proporcionan la comodidad de los miembros de datos públicos sin los riesgos que provienen del acceso sin comprobar, sin controlar y sin proteger a los datos del objeto.
- **Modificadores de acceso:** La forma que los programas orientados a objetos, provee para que un programador obligue a respetar el Principio de Ocultamiento son los llamados modificadores de acceso. Se trata de ciertas palabras reservadas que colocadas delante de la declaración de un atributo o de un método de una clase, hacen que ese atributo o ese método tengan accesibilidad más amplia o menos amplia desde algún método que no esté en la clase. Así, los modificadores de acceso pueden ser: public, private, protected
 - **Public:** un miembro público es accesible tanto desde el interior de la clase (por sus propios métodos), como desde el exterior de la misma (por métodos de otras clases).
 - **Private:** sólo es accesible desde el interior de la propia clase, usando sus propios métodos.
 - **Readonly:** El acceso es de sólo lectura.
 - **Protected:** aplicable en contextos de herencia (tema que veremos más adelante), hace que un miembro sea público para sus clases derivadas y para clases en el mismo paquete, pero los hace privados para el resto.

Instancias

Para manipular los objetos o instancias de las clases (tipos) también se utilizan variables y éstas tienen una semántica propia la cual, se diferencia de los tipos básicos. Para ello, deberemos usar explícitamente el operador NEW. En caso contrario contendrán una referencia a null, lo que semánticamente significa que no está haciendo referencia a ningún objeto.

Sintaxis para instanciar objetos:

<nombre_objeto> = new <Nombre_de_Clase>(<parámetros>)

Hay 3 maneras de inicializar un objeto. Es decir, proporcionar datos a un objeto.

1. Por referencias a variables.

Ejemplo:

```
let persona= new Persona();
persona.apellido="Rosas";
persona.nombre ="Maria";
```

2. Por medio del constructor de la clase.

Ejemplo:

```
let persona= new Persona ("Maria", "Rosas");
```

3. Por medio de la propiedad setter.

```
let persona= new Persona();
persona.Apellido="Rosas";
persona.Nombre ="Maria";
```

Recomendaciones

Aunque cada programador puede definir su propio estilo de programación, una buena práctica es seguir el estilo utilizado por los diseñadores del lenguaje pues, de seguir esta práctica será mucho más fácil analizar el fuente de terceros y, a su vez, que otros programadores analicen y comprendan nuestro fuente.

- Utilizar identificadores nemotécnicos, es decir, utilizar nombres simbólicos adecuados para los identificadores lo suficientemente autoexplicativos por sí mismos para dar una orientación de su uso o funcionalidad de manera tal que podamos hacer más claros y legibles nuestros códigos.
- Los identificadores de clases, módulos, interfaces y enumeraciones deberán usar PascalCase.
- Los identificadores de objetos, métodos, instancias, constantes y propiedades de los objetos deberán usar camelCase.
- Utilizar comentarios, pero éstos seguirán un formato general de fácil portabilidad y que no incluya líneas completas de caracteres repetidos. Los que se coloquen dentro de bloques de código deben aparecer en una línea independiente indentada de igual forma que el bloque de código que describen.
- Evitar en lo posible líneas de longitud superior a 80 caracteres.
- Indentar los bloques de código.

Dejamos para su lectura un documento con los conceptos ya vistos en la guía anterior, ampliando teoría:

[POO en TypeScript](#)

12. Referencias

- CampusMPV. TypeScript contra JavaScript: ¿cuál deberías utilizar?. <https://www.campusmvp.es/recursos/post/typescript-contra-javascript-cual-deberias-utilizar.aspx>
- Strephonsays. Diferencia entre JavaScript y TypeScript. <https://es.strephonsays.com/javascript-and-vs-typescript-13697>
- Microsoft. Declaración de clases en TypeScript y creación de una instancia de estas. <https://docs.microsoft.com/es-es/learn/modules/typescript-declare-instantiate-classes/>
- Felipe Steffolani (2020) Apuntes Programa Clip.
- Laura Álvarez, Helmer Avendaño, Yeison García, Sebastián Morales, Edwin Bohórquez, Santiago Hernandez, Sebastián Moreno, Cristian Orjuela. Programación Orientada a Objetos, https://ferestrepoca.github.io/paradigmas-de-programacion/poo/poo_teor%C3%ADa/concepts.html
- 111Mil. Módulo Programación Orientada a Objetos. <https://github.com/111milprogramadores/apuntes/blob/master/Programacion%20Orientada%20a%20Objetos/Apuntes%20Teoricos%20de%20Programacion>
- <https://www.typescriptlang.org/>