TypeScript

**DOCUMENTO DE USO INTERNO**

La información contenida en este documento ha sido clasificada como de uso interno y, por tanto, usted deberá ser empleado de Helicon para manejar el presente documento. En caso contrario, favor de entregar este documento a cualquier empleado de Helicon. En caso de ignorar este aviso estará sujeto a las medidas legales aplicables.

Este documento es para uso exclusivo del personal de **Helicon.** Ninguna de sus partes puede ser circulada, citada o reproducida para distribución fuera de las instalaciones **Helicon.**

**HISTORIAL DE VERSIONES**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Vers.** | **Descripción** | **Elaborado por:** | **Revisado por:** | **Autorizado por:** |
| 1.0 | Creación del documento para el proceso. | Nombre: Osvaldo Flores. | Nombre: Angelo Raimondi. | Nombre: Angelo Raimondi |
|  |  |  |  |  |

# Tipos de datos

TypeScript es conocido por ser un lenguaje fuerte en cuanto a tipos de datos. TypeScript soporta los mismos tipos de datos que JavaScript, pero con algunas cosas extra.

## Boolean

Solo puede ser asignado el valor de **true** o **false**, los cuales son reconocidos como valores *booleanos*

**const** isGreen: boolean = **true**;

## Number

Tanto en JavaScript como en TypeScript, todos los números son valores punto flotante. TypeScript soporta el uso de hexadecimal, decimal, octal y binario.

**const** age: number = 21;

## String

TypeScript utiliza comillas dobles o comillas simples para manejar cadenas de caracteres. Se pueden escribir strings de múltiples líneas. Así mismo se puede interpolar strings con expresiones mediante el uso de comilla inversa (**`**).

**const** greetting: string = `Hello, I am **${age}** years old`;

## List

TypeScript puede trabajar con arreglos, es necesario especificar el tipo y existen dos maneras de hacerlos.

Especificando el tipo de dato:

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

O de manera genérica y especificando el tipo de dato:

**let** list: Array<number> = [1, 2, 3];

## Tupla

Las tuplas permiten hacer un arreglo con un número fijo de elementos, los tipos de datos de los elementos tienen que conocerse, pero no es necesario que sean los mismos.

**const** user: [string, int] = ["Osvaldo", 21];

## Enum

Es una manera más sencilla de nombrar un conjunto de valores numéricos. Dichos valores empezaran con un índice de 0, sin embargo, este puede ser modificado. Un **Enum** puede ser accedido mediante su índice o su nombre:

**enum** Color{ Red, Blue, Green }; //index: 0, 1, 2.

**const** myColor: Color = Color.Green;

console.log(myColor); // Output: 0.

## Any

Una manera de trabajar con valores dinámicos es utilizar **any**, así mismo puede ser utilizado en *genéricos* para poder hacer listas dinámicas.

**let** something: any = 'Hello World';

something = 99;

## Void

Sirve para indicar la ausencia de cualquier tipo de dato. Comúnmente se utiliza para especificar que una función no retornará algo.

## Undefined y Null

Por default son subtipos de todos los otros tipos, es decir podemos asignar **null** y **undefined** a cualquier tipo de dato.

## Object

Los objetos representan un tipo de dato no-primitivo. En TypeScript están compuestos por el par **key-value**.

**let** newUser: any = {

name: 'Carlos',

age: 22,

};

## Union

Las variables no están restringidas a solo ser asignadas a un tipo de dato. Con **unión** *podemos asignar dos o más* tipos de datos a una variable.

**let** number: number | string = 25;

number = '24';

# Funciones

Las funciones en TypeScript trabajan de la misma manera que en JavaScript, pero con algunas nuevas características.

## Tipos de dato en funciones

Las funciones *recibirán parámetros* especificando el **tipo de dato** de cada parámetro. Así mismo *retornarán un valor* especificando el tipo de dato de retorno.

Una función que no retorna nada, será del tipo de dato **void**. Las variables no suelen ser declaradas con este tipo de dato, es preferible utilizar (**undefined** o **null**).

## Parámetros opcionales

Los parámetros de las funciones pueden ser *opcionales* agregando un (**?**) al final del nombre del parámetro. El valor del parámetro en caso de no ser enviado será **undefined**.

*Los parámetros opcionales deben de ir después de los* ***requeridos****.*

Los parámetros pueden tener valores *default*, si asignamos **undefined** o no pasamos nada a dicho parámetro, se asignará el valor default.

# Clases

Al igual que en JavaScript, se utilizan clases con modelos parecidos al de *java*. Tradicionalmente se trabajaba con funciones y herencia basada en prototipos.

Las clases suelen tener métodos y propiedades y podemos inicializar dichas propiedades con un **constructor**. El constructor se ejecutará cuando se mande a llamar (creando una instancia).

TypeScript ofrece que las propiedades y métodos de una clase tengan modificadores de acceso como: **private, public, proctected.** En TypeScript los miembros son públicos por default.

## Miembros privados

Cuando un miembro es marcado como **private**, este no puede ser accedido por fuera de la clase. Para hacer referencia a un miembro dentro de una clase utilizaremos la palabra **this**.

**private** aNumber: int;

## Miembros protegidos

Un miembro protegido en TypeScript solo puede ser accedido desde su propia clase o desde una superclase (heredando).

**protected** name: string;

## Miembros de solo lectura

Así mismo se pueden hacer propiedades de solo lectura con la palabra reservada **readonly**. Los miembros de solo lectura pueden ser definidos desde el constructor.

**readonly** constantValue: int;

## Herencia

Las clases en TypeScript pueden ser heredadas; declarar otra clase que herede todos los miembros de la clase padre. La palabra reservada **super** llama al constructor de la clase padre y ejecuta la lógica que esté dentro.

## Get y Set

TypeScript soporta el uso de **get** y **set**, pueden ser utilizados para crear las funciones getter y setter las cuales retornarán o modificarán un miembro.

## Static

Las propiedades y métodos static son miembros que pueden ser accedidos sin la necesidad de instanciar una clase.

**static** PI: number = 3.1415;

## Clases abstractas

Se utiliza la palabra reservada **abstract** antes de **class** para crear una clase abstracta. Puede contener métodos abstractos los cuales son vacíos.

Una clase abstracta sirve como un molde para definir como queremos que sea nuestra clase, dichas clases abstractas no pueden instancias objetos solo pueden **heredar**.

# Interfaces

En una **interfaz** se puede definir propiedades y métodos, para más adelante llamar a dicha interfaz y tratarla como un *“tipo de dato”.*

Podemos definir una interfaz escribiendo **interface** *seguido del nombre* de dicha interfaz *utilizando* **UpperCamelCase**, definiendo sus propiedades y métodos dentro del bloque.

## Implementación de interfaces

En adición a comportarse como un *“tipo de dato”*, las interfaces pueden ser **implementadas** en clases, generando una clase de *“contrato”* en donde todos los propiedades y métodos de la interfaz necesitan ser *implementados* en la clase.

## Herencia de interfaces

Al igual que las clases, las interfaces pueden heredar sus propias propiedades a otras interfaces con la palabra reservada **extends**.

## Propiedades opcionales

Se pueden definir propiedades de manera opcional agregando un **(?)** después del nombre de la propiedad.

somethingOptional?: string;

## Propiedades de solo lectura

Algunas propiedades se les puede asignar un valor solo una vez y tratarla solo para lectura (sin poder modificarla), especificando **readonly** antes del nombre de la propiedad. Igualmente cuenta con **ReadonlyArray<T>**.

**readonly** x: number;

## Interfaz para funciones

Podemos crear una interfaz para diseñar funciones con ciertos parámetros y que regrese un tipo en específico.

# Genéricos

Los genéricos son una manera de escribir código dinámico y flexible, permitiéndonos utilizar distintos tipos de datos.

Una *manera simple* de hacer algo *genérico* es utilizar **any**, de esta manera aceptará cualquier tipo de dato. Sin embargo, esto no es practico porque se desconocerá con que tipo de dato se está trabajando.

La manera de trabajar con genéricos es utilizando **<T>** en donde **T** es el tipo de dato por trabajar.