**Tecnologías: Desarrollo Web. Typescript**

**https://camo.githubusercontent.com/fe76a161cd73881d4f804eb1bb6bccd54a125fe141ebad3bf8910aea4d584fc6/68747470733a2f2f692e696d6775722e636f6d2f534757594d31632e706e67 Introducción**

**JavaScript** es un lenguaje desarrollado para ser utilizado principalmente en el desarrollo de aplicaciones web dinámicas. Más recientemente, también se ha popularizado como un lenguaje para desarrollar aplicaciones *back-end* utilizando [node.js](https://nodejs.org/es/), que es un intérprete de JavaScript del lado del servidor.

JavaScript es un lenguaje no tipado, de scripts y funcional. Los scripts se embeben en una página Web y las instrucciones de ese script las analiza y procesa (o *interpreta*) el navegador en el momento que deben ser ejecutadas. El hecho de que los programas escritos en JavaScript no se compilen tiene como consecuencia que muchos de los errores que puede cometer el programador, solo se van a detectar en tiempo de ejecución. Esto es más evidente con los errores relacionados con los tipos de datos, por ejemplo: valores incompatibles entre las variables cuando estas se operan; cambios de tipo en las variables; funciones con parámetros que no son recibidos; valores de retorno de las funciones que no son los esperados, entre otros.

Una alternativa para solucionar estos problemas es *TypeScript*. Este es un lenguaje de programación, creado por Microsoft, que amplía JavaScript. El lenguaje se ha creado como un proyecto de código abierto, con licencia de Apache License 2.0, por lo que la comunidad de desarrolladores puede usarlo libremente. Entre sus características más notables, es que TypeScript brinda seguridad de tipos de datos a JavaScript. Esta característica permite a los desarrolladores ser muy efectivos e introducir menos errores gracias a que los programas escritos en TypeScript se compilan.

El resultado de compilar un programa TypeScript es obtener su equivalente en JavaScript. Adicionalmente, en este proceso se detectan potenciales problemas del código que el desarrollador puede corregir antes de liberar el JavaScript que será utilizado en el desarrollo de su aplicación web.

TypeScript es un superconjunto de JavaScript que incluye conceptos de tipos de datos, clases y módulos.

| **Figura 1** |
| --- |
| Figura 1: TypeScript se compila a JavaScript |

Recomendamos visitar el siguiente link donde podrá encontrar un [tutorial básico de Typescript](https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html). En este capítulo mencionaremos algunos conceptos importantes, pero no intenta ser exhaustivo con respecto al lenguaje.

**https://camo.githubusercontent.com/fe76a161cd73881d4f804eb1bb6bccd54a125fe141ebad3bf8910aea4d584fc6/68747470733a2f2f692e696d6775722e636f6d2f534757594d31632e706e67 Tipos básicos de datos en TypeScript**

Los programas escritos en lenguajes con tipos de datos son más fáciles de entender y de corregir. El compilador puede encontrar errores antes de ejecutar el código y esto hace el proceso de desarrollo más productivo.

TypeScript provee un sistema de tipos de datos básicos como booleanos, números y strings. También provee constructores para enumeraciones, arreglos, tuplas y también clases e interfaces para que el desarrollador defina sus propios tipos. De la misma forma que se hace en JavaScript, es posible escribir programas en TypeScript sin proveer en las variables y/o los parámetros los tipos de datos. En ese caso hablamos de tipos de datos **implícitos**: el tipo de la variable será el correspondiente al valor que se asigne la primera vez. Sin embargo, nosotros siempre usaremos el sistema de tipos de datos **explícito** y en modo estricto, para apoyarnos en el compilador y las herramientas de edición; esto con el fin de realizar una detección temprana de defectos en el código.

En este capítulo veremos los tipos de datos básicos en TypeScript primero asociados con las variables y luego asociados con los parámetros de las funciones. En el próximo veremos clases e interfaces para entender cómo el desarrollador puede definir sus propios tipos de datos.

**Variables**

Las palabras reservadas para definir variables en TypeScript son **const** y **let**. El nombre de una variable puede ser una combinación de letras, dígitos (no puede empezar por un dígito), el símbolo "\_" y el símbolo "$".

No se puede hablar de variables en TypeScript sin hablar primero de cómo se definen en JavaScript. Recordemos que TypeScript es un superconjunto de JavaScript, es decir, lo que es válido en JavaScript también debe serlo en TypeScript. Por ejemplo, en JavaScript, la palabra reservada **var** permite definir una variable cuyo alcance es el de la función donde se definió. Por ejemplo, en el siguiente código tenemos que:

var ejemplo = 123;

if (true) {

var ejemplo = 456;

}

Al salir de este fragmento de código, la variable ejemplo tiene por valor 456. Aun cuando se está declarando una nueva variable dentro de un bloque de control {. ..} en la línea 3, su valor prevalece después de terminado este bloque.

Al considerar que este mecanismo propicia que el desarrollador se equivoque, en TypeScript se tiene la palabra reservada **let** con la que se define una variable cuyo alcance es el bloque de control:

let ejemplo: number = 123;

if (true) {

let ejemplo: number = 456;

}

Al salir de este fragmento de código, la variable ejemplo tiene por valor 123.

Por otro lado, para definir constantes o variables \_ inmutables \_, se utiliza la palabra reservada **const**. Por ejemplo:

const maxRows = 10;

if (x > maxRows) {

}

La variable maxRows definida como const debe tener obligatoriamente un valor inicial que no cambiará en el código donde esta variable tiene alcance.

**El tipo de dato any**

El tipo de dato "any", en realidad significa que no sabemos cuál es el tipo de dato de la variable y que no queremos que el compilador haga validaciones de tipo. Por ejemplo:

let variablesintipo: any = 10;

variablesintipo = "esto es una cadena";

variablesintipo = false;

Esto puede servir para trabajar con código javascript ya existente, pero en la medida de lo posible siempre trataremos de definir el tipo de las variables a menos de que tengamos una razón clara para no hacerlo.

**Booleanos, Números y Strings**

Los tipos básicos en TypeScript pueden ser booleanos cuyo valor es true o false, numéricos cuyo valor es un número ya sea entero o flotante y cadenas de caracteres. Los siguientes ejemplos ilustran su uso:

let numerico: number;

numerico = 111;

numerico = 111.234;

numerico = '111'; // Error

let cadena: string;

cadena = '123';

cadena = 123; // Error

let opcion: boolean;

opcion = true;

opcion = false;

opcion = 'false'; // Error

En las líneas 4, 9 y 15 se puede ver lo que hace el compilador gracias al sistema de tipos: identifica errores de incongruencia de los valores asignados a las variables y el tipo con el que fueron declaradas.

Para ver los operadores disponibles sobre estos tipos de datos puede acceder al [enlace](https://www.tutorialspoint.com/typescript/typescript_operators.htm).

**Void, Null, Undefined**

void representa la ausencia de valor. Se utiliza cuando una función no retorna nada. null y undefined son tipos de datos (subtipos de todos los demás) y también son valores.

Por ejemplo, a una variable declara como string se le puede asignar el valor null o undefined.

**https://camo.githubusercontent.com/fe76a161cd73881d4f804eb1bb6bccd54a125fe141ebad3bf8910aea4d584fc6/68747470733a2f2f692e696d6775722e636f6d2f534757594d31632e706e67 Enumeraciones, Arreglos, Tuplas**

**Enumeraciones**

Las enumeraciones nos permiten definir un conjunto de constantes nombradas. Por ejemplo,

enum TipoCurso {

CBU = 0,

Nivel1 = 1,

Profesional = 2,

Postgrado = 3

}

let micurso: TipoCurso = TipoCurso.Profesional;

Más sobre enumeraciones [Typescript Hanbook\_enum](https://www.typescriptlang.org/docs/handbook/enums.html)

**Arreglos**

En TypeScript, como JavaScript, podemos definir arreglos de valores. Los tipos de arreglos se pueden escribir de dos Formas:

En la primera, usa el tipo de los elementos seguido de [] para denotar un arreglo de ese tipo:

let listanumeros: number [] = [1, 2, 3];

En la segunda, se utiliza la palabra reservada Array y se indica el tipo de los elementos:

let list: Array <número> = [1, 2, 3]

**Tuplas**

Las tuplas tienen un número predefinido de elementos, pero éstos pueden ser de tipos de dato distintos. Por ejemplo:

let unatupla: [string, number];

unatupla = ["texto", 100];

**Arreglo de string y un ciclo**

De manera general un arreglo se puede declarar:

var nombrearreglo: tipoelementos []

Y para inicializar:

nombrearreglo = [elemento1, elemento2, ...]

Por ejemplo, tenemos la declaración del arreglo de string llamado lista y al mismo tiempo su inicialización.

En el siguiente ejemplo, tenemos un ciclo estándar sobre el arreglo que lo recorre desde la posición 0 hasta su longitud:

var list :string[] = ["juan","pedro","maria","luisa"];

function concatenateNames(list: string[]) {

let concatenatedNames = "";

for (let index = 0; index < list.length; index++) {

concatenatedNames += list[index] + "";

}

return concatenatedNames;

}

El siguiente ejemplo es una variante del anterior en donde se utiliza forEach para aplicar una función sobre cada uno de los elementos de la lista:

function concatenateNamesForEach(list: string[]) {

let concatenatedNames = "";

list.forEach(function(value) {concatenatedNames += value +" "}) ;

return concatenatedNames;

}

console.log(concatenateNamesForEach(list));

La función dentro del forEach también puede definirse con el operador =>

function concatenateNamesForEach(list: string[]) {

let concatenatedNames = "";

list.forEach(value => {concatenatedNames += value +" "}) ;

return concatenatedNames;

}

console.log(concatenateNamesForEach(list));

El siguiente ejemplo es otra variante en donde se utiliza map para aplicar una función definida con el operador => sobre cada uno de los elementos de la lista:

function concatenateNamesMap(list: string[]) {

let concatenatedNames = "";

list.map(value => {concatenatedNames += value +" "});

return concatenatedNames;

}

console.log(concatenateNamesMap(list));

Otra manera es utilizando la función reduce la cual invoca una función "reducidora" por cada uno de los elementos, las funciones reducidoras pueden llegar a tener 4 parámetros:

* acumulador
* valoractual
* indiceactual
* arreglo

El siguiente ejemplo es una implementación de la concatenación de un arreglo utilizando reduce con un acumulador y el valor actual:

function concatenateNamesReduce(list: string[]) {

return list.reduce(conc);

}

function conc(concatenatedNames, valor) {

return concatenatedNames + " " + valor;

}

console.log(concatenateNamesReduce(list));

La función dentro del reduce también puede definirse con el operador =>

function concatenateNamesReduce(list: string[]) {

return list.reduce((concatenatedNames, valor) => concatenatedNames + " " + valor);

}

console.log(concatenateNamesReduce(list));

**https://camo.githubusercontent.com/d415e53fcf416a5f3b7a2f84d470d3ac4afa1a670a8d925a20e4ade513e7f9e6/68747470733a2f2f692e696d6775722e636f6d2f54784d6a5442662e706e67 Funciones y Clases**

**https://camo.githubusercontent.com/fe76a161cd73881d4f804eb1bb6bccd54a125fe141ebad3bf8910aea4d584fc6/68747470733a2f2f692e696d6775722e636f6d2f534757594d31632e706e67 Funciones**

**Declaración de una función simple**

En el siguiente código declaramos la función sumar. Está definida con dos parámetros de tipo number y retorna un number.

function sumar(x: number, y: number): number {

return x+y;

}

console.log(sumar(5,5))

Utilizando el mismo ejemplo, para ilustrar el sistema de tipos, vamos a cambiar la función para que devuelva el resultado de la suma en un string. Note que estamos usando la palabra reservada let para definir una variable local, de tipo number, a la función. También usamos el método toString(), predefinido para las variables de tipo number

function sumar(x: number, y: number): string {

let result: number = x+y;

return result.toString();

}

console.log(sumar(5,5))

**Función anónima**

Las funciones anónimas son funciones que no tienen nombre. En el siguiente ejemplo, una función anónima se asigna a una variable llamada msg. Luego la variable se utiliza para invocar la función.

var msg = function() {

return "Hola Mundo";

}

console.log(msg())

**Función módulo y parámetro opcional**

Una función puede definir parámetros opcionales, es decir que, al llamar la función, puede no estar el parámetro. En el siguiente ejemplo, el parámetro opcional es módulo. Se indica que es opcional con el símbolo ?.

El ejemplo también muestra el uso del operador módulo: %. Si no se da el valor del módulo se devuelve el residuo de dividir por 2.

function fmodulo(x: number, modulo?: number): number {

if (modulo && modulo > 0) {

return x % modulo;

} else {

return x % 2;

}

}

console.log(fmodulo(35, 30));

console.log(fmodulo(35));

**Funciones como objetos**

En TypeScript/JavaScript las funciones también son objetos. En los siguientes ejemplos podemos ver algunas de las funcionalidades que esto nos trae.

**Funciones que retornan una función**

En este primer la función hola imprime `Hola` y luego retorna la función mundo.

function hola() {

console.log('Hola');

function mundo(){

console.log(' Mundo!');

}

return mundo;

}

var s = hola();

console.log('mundo: ' + s);

Al ejecutarlo se imprime lo siguiente en consola:  
Hola  
mundo: function mundo(){ console.log(' Mundo!'); }  
En el siguiente ejemplo se ejecuta la función mundo que retorna la función hola:

function hola() {

console.log('Hola');

function mundo(){

console.log(' Mundo!');

}

return mundo;

}

var s = hola();

console.log('break');

s();

En consola:  
Hola  
Break  
Mundo

| **Pregunta: ¿Qué sale en consola al ejecutar el siguiente código?** |
| --- |

function hola() {

console.log('Hola');

function mundo(){

console.log(' Mundo!');

}

return mundo();

}

var s = hola();

console.log('break');

s();

**Funciones por parámetro**

De la misma manera las funciones también se pueden enviar por parámetro:

function hola() {

console.log('Hola');

function mundo(){

console.log(' Mundo!');

}

return mundo;

}

function funcionPorParametro(f : function){

f()

f()()

}

funcionPorParametro(hola);

En consola:

Hola  
Mundo

**https://camo.githubusercontent.com/fe76a161cd73881d4f804eb1bb6bccd54a125fe141ebad3bf8910aea4d584fc6/68747470733a2f2f692e696d6775722e636f6d2f534757594d31632e706e67 Clases e Interfaces TypeScript**

JavaScript a partir de ECMAScript2015 introdujo clases al lenguaje. Sin embargo, es importante tener en cuenta que estas clases no cambian la herencia basada en prototipos del lenguaje. TypeScript permite definir clases antes de que sean *transpiladas* a JavaScript ES6. Por lo tanto, los desarrolladores pueden extender clases para construir nuevas clases usando herencia.

Para este ejemplo instanciamos una universidad empezando por la clase Student que hereda de la clase Person.

class Person {

name: string;

code: number;

cardId: number;

constructor(completeName: string,code: number, cardId: number) {

this.name = completeName;

this.code = code;

this.cardId = cardId;

}

}

class Student extends Person {

age: number;

address: string;

phone:string;

currentCourses: Course[];

constructor(completeName: string,code: number, cardId: number, age:number,address:string,phone:string) {

super(completeName,code,cardId);

this.age = age;

this.address = address;

this.phone = phone;

}

}

Como se puede observar la sintaxis es similar a Java o C#. En las dos clases anteriores hemos definido atributos, con su respectivo tipo de datos, y el constructor de la clase. Note que en el constructor de la clase estamos usando this para hacer referencia al objeto actual.

Observe que la clase Student declara el atributo currentCourses de tipo array de Course definida como sigue:

class Course {

name: string;

credits: number;

teacher: Teacher;

constructor(name: string,credits:number, teacher:Teacher ) {

this.name = name;

this.credits = credits;

this.teacher = teacher;

}

}

| **Ver en**[**StackBlitz**](https://stackblitz.com/edit/clases-e-interfaces-v1) |
| --- |

**Construcción de objetos**

En el [proyecto](https://stackblitz.com/edit/clases-e-interfaces-v2) aparecen dos archivos (y dataStudent y dataCourses). El archivo dataStudent contiene los datos de una estudiante y dataCourses contiene un arreglo con sus cursos. Para inicializar algunos objetos de nuestras clases y hacer más ejemplos, vamos a construir una función para leer los datos de un arreglo de objetos de tipo Course y asociarlos con la estudiante.

En el index.ts tenemos el siguiente código:

var student: Student = dataStudent;

function cargarDatos() {

dataCourses.map(function (course) {

student.currentCourses.push(course);

});

}

cargarDatos();

console.log(student);

Recordemos que la función map se ejecuta sobre un arreglo ([map](https://www.tutorialspoint.com/typescript/typescript_array_map.htm)). En este caso sobre el arreglo que contiene la lista de cursos dataCourses. La función map ejecuta sobre cada elemento del arreglo la función que va de parámetro:

function (course) {

student.currentCourses.push(course);

}

La función recibe el curso y lo que hace es agregarlo al atributo currentCourses de la estudiante. Esto funciona ya que JavaScript automáticamente hace el equivalente a un Cast en java cuando se asigna la constante dataStudent a una variable de tipo Student e igualmente con los cursos.

Ver en [StackBlitz](https://stackblitz.com/edit/clases-e-interfaces-v2)

**Interfaces**

Las interfaces en JavaScript, como en Java, son colecciones de signaturas de métodos y atributos que especifican qué se debe hacer, pero no su implementación. Dado que JavaScript es un lenguaje dinámico, es muy poca la utilidad de interfaces en el momento de programar. Sin embargo, el tipado estático de TypeScript las hace de toda utilidad porque va a forzar la verificación de tipos: las decisiones tomadas en la interfaz se deben cumplir. Sigamos con el ejemplo y veamos la definición de la interfaz Teacher:

interface Teacher extends Person {

numOffice: string;

[propName: string]: any;

}

El atributo [propName: string]: any permite que un objeto Teacher tenga múltiples atributos, de tipo string, cuyo nombre puede variar en la interfaz . Para este ejemplo se les definió un atributo University en el [archivo](https://stackblitz.com/edit/clases-e-interfaces?file=dataCourses.ts) de la información dummy de los cursos.

De esta manera cuando el index.ts imprime la información del estudiante, sus cursos y los respectivos profesores, los objetos Teacher contienen el atributo University definido. Si bien esta facilidad es interesante, no es recomendada, dado que lo que queremos es evitar introducir errores y el que el compilador revise los tipos es una gran ayuda.

| **Pregunta: Qué pasa si se le intenta asignar un atributo de tipo *numero* a uno de los profesores desde el archivo de carga?** |
| --- |

Ver en [StackBlitz](https://stackblitz.com/edit/clases-e-interfaces)

**https://camo.githubusercontent.com/d415e53fcf416a5f3b7a2f84d470d3ac4afa1a670a8d925a20e4ade513e7f9e6/68747470733a2f2f692e696d6775722e636f6d2f54784d6a5442662e706e67 Tutorial TypeScript**

**Objetivos**

Al finalizar este tutorial el estudiante estará en capacidad de:

* Construir un sitio web responsive usando Html y CSS.
* Utilizar el framework Bootstrap para la maquetación de sitios web.
* Construir funciones en typescript.

**Contexto**

Construcción de la hoja de vida del estudiante, es decir, su hoja de vida. Esta página de hoja de vida debe contener la imagen del estudiante, sus datos personales y los cursos que está tomando actualmente. La siguiente figura muestra un mockup final de la hoja de vida del estudiante.

| **Imgur** |
| --- |
| Figura 1: Mockup Hoja de vida del Estudiante |

**https://camo.githubusercontent.com/fe76a161cd73881d4f804eb1bb6bccd54a125fe141ebad3bf8910aea4d584fc6/68747470733a2f2f692e696d6775722e636f6d2f534757594d31632e706e67 Paso 1**

Vamos a utilizar VSCode para desarrollar este Tutorial.

**Paso 1.1.** Cree en su máquina un folder llamado tutorialTypescript.

**Paso 1.2.** Abra ese folder en VSCode

**Paso 1.3.** Desde la terminal inicie un proyecto TypeScript con el comando tsc --init

**Paso 1.4.** Abra el archivo tsconfig.jsony asegúrese de que las siguientes opciones estén descomentadas y con los valores correctos

"target": "es5",

"module": "es6",

"outDir": "./scripts",

"rootDir": "./",

"strict": true,

"alwaysStrict": true,

"esModuleInterop": true,

"forceConsistentCasingInFileNames": true

**Paso 1.5.** Cree dentro de la carpeta un archivo denominado index.html

Construcción inicial de la hoja de vida del estudiante.

Para este paso vamos a utilizar sólo HTML. Por lo tanto, todos los elementos agregados estarán ubicados uno tras otro a modo de lista.

**Paso 1.6.** Copie el siguiente código en el archivo index.html.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Tutorial 1</title>

</head>

<body>

<h1>Ana María Suarez</h1>

<img src="./avatar.png" height="300px" width="300px" alt="Avatar">

<h2>Datos Personales</h2>

<table>

<thead>

<tr>

<th></th>

</tr>

</thead>

<tbody>

<tr>

<td>Código</td>

<td>10234320</td>

</tr>

<tr>

<td>Cédula</td>

<td>12221122</td>

</tr>

<tr>

<td>Edad</td>

<td>25 Años</td>

</tr>

<tr>

<td>Dirección</td>

<td>Calle falsa 123</td>

</tr>

<tr>

<td>Teléfono</td>

<td>018002500</td>

</tr>

</tbody>

</table>

<h2>Cursos Actuales</h2>

<table>

<thead>

<tr>

<th>Curso</th>

<th>Profesor</th>

</tr>

</thead>

<tbody>

<tr>

<td>Ingeniería de SW</td>

<td>Pablo Picasso</td>

</tr>

<tr>

<td>Futbol I</td>

<td>Juan Piña</td>

</tr>

<tr>

<td>Algoritmos Genéticos</td>

<td>María de la Rosa</td>

</tr>

</tbody>

</table>

</body>

</html>

**Paso 1.7.** Salve su archivo index.html y abralo con Open Live Server ([Instrucciones live server](https://github.com/Uniandes-isis2603/recursos-isis2603/wiki/visualcode))

**Solución**

En la figura 2 tenemos el despliegue de una solución.

\*\*Paso 1.8. \*\* Ud. debe cambiar por su información, incluyendo su imagen o su avatar, el contenido del index.html.

| **Imgur** |
| --- |
| Figura 2: Paso 1 |

**https://camo.githubusercontent.com/fe76a161cd73881d4f804eb1bb6bccd54a125fe141ebad3bf8910aea4d584fc6/68747470733a2f2f692e696d6775722e636f6d2f534757594d31632e706e67 Paso 2**

En este paso vamos a utilizar Bootstrap en la página creada en Paso 1.

**Paso 2.1.** Cambie la etiqueta head de su página por lo siguiente para incluir Bootstrap.

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Tutorial 1</title>

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

</head>

**Paso 2.2.** Salve el archivo y revise el despliegue de la página para ver los cambios con los estilos bootstrap (al menos el tipo de letra).

Una vez incluido Bootstrap en el sitio web vamos a establecer la estructura visual del sitio. Como vemos en el mockup existen dos columnas y cada una tiene un título y una tabla. Bootstrap permite la distribución de elementos mediante el uso de los estilosrowy col, las cuales definen la ubicación de los elementos en el DOM.

Definamos un contenedor y dentro de éste una fila que contiene dos columnas.

| **Imgur** |
| --- |
| Figura 3: Contenedor para la Hoja de vida del Estudiante |

Lo cual en Bootstrap equivale a:

<body>

<div class="container">

<div class="row">

<div class="col">

<h1>Ana María Suarez</h1>

<img src="./avatar.png" alt="Avatar">

<h2>Datos Personales</h2>

<table>

<thead>

<tr>

<th></th>

</tr>

</thead>

<tbody>

<tr>

<td>Código</td>

<td>10234320</td>

</tr>

<tr>

<td>Cédula</td>

<td>12221122</td>

</tr>

<tr>

<td>Edad</td>

<td>25 Años</td>

</tr>

<tr>

<td>Dirección</td>

<td>Calle falsa 123</td>

</tr>

<tr>

<td>Teléfono</td>

<td>018002500</td>

</tr>

</tbody>

</table>

</div>

<div class="col">

<h2>Cursos Actuales</h2>

<table>

<thead>

<tr>

<th>Curso</th>

<th>Profesor</th>

</tr>

</thead>

<tbody>

<tr>

<td>Ingeniería de SW</td>

<td>Pablo Picasso</td>

</tr>

<tr>

<td>Futbol I</td>

<td>Juan Piña</td>

</tr>

<tr>

<td>Algoritmos Genéticos</td>

<td>María de la Rosa</td>

</tr>

</tbody>

</table>

</div>

</div>

</div>

</body>

**Paso 2.2.** Organice la grilla de su página. Para esto defina las etiquetas div y los estilos, container, row, col, necesarios para lograr la maquetación del ejemplo o una que Ud. prefiera. Tenga cuidado cuando agregue etiquetas para que abran y cierren en los lugares correctos. Formatee el documento para que pueda verificar que las etiquetas se abren y cierran donde se necesita.

**Paso 2.3.** Cómo quedó la imagen en su página? Para lograr que la imagen sea responsiva, utilice los siguientes estilos de Bootstrap:

<img class="img-fluid max-width: 100% height: auto" src="./avatar.png" alt="Avatar">`

**Desafíos**

**Paso 2.4.** Mejore su página teniendo en cuenta:

* Definir márgenes y paddings para que los elementos se vean mejor en la página.
* Agregar clases de Bootstrap a las Tablas y agregar el encabezado de la tabla **Cursos Actuales** en Negro.
* Agregar la clase Card a las tablas de tal forma que se vea la separación de las secciones.

**Solución simple**

Aquí una posible solución.

| **Imgur** |
| --- |
| Figura 4: Paso 2 |

**https://camo.githubusercontent.com/fe76a161cd73881d4f804eb1bb6bccd54a125fe141ebad3bf8910aea4d584fc6/68747470733a2f2f692e696d6775722e636f6d2f534757594d31632e706e67 Paso 3**

En este paso vamos a escribir código en TypeScript para:

* Modelar los cursos como objetos y crearlos.
* Iterar sobre ellos para desplegarlos sobre el HTML en la tabla.
* Crear un filtro en la tabla.
* Mejorar la apariencia utilizando más estilos de Bootstrap.

**Definición de las clases y los objetos**

Vamos a crear la clase Course para modelar un curso. Esta clase la creamos en un archivo llamado course.ts.

export class Course {

name: string;

credits: number;

professor: string;

constructor(name: string, professor: string, credits: number) {

this.name = name;

this.credits = credits;

this.professor = professor;

}

}

Adicionalmente, tenemos un arreglo llamado dataCourses en donde definimos los objetos Course que mostraremos en la página.

Este arreglo lo declaramos en un archivo llamado dataCourses.ts y su contenido es:

import { Course } from './course.js';

export const dataCourses = [

new Course("Ingeniería de Sw", "Pablo Picasso", 4),

new Course("Futbol 1", "Freddy Rincón", 2),

new Course("Algoritmos", "Carlos Fuentes", 2),

new Course("Estructuras de Datos", "Yesid D", 1),

new Course("Futbol 2", "James R", 6)

]

**Modificaciones al HTML**

El objetivo de este paso es remplazar el código HTML de la tabla de cursos actuales, cableado dentro del HTML, por el llamado a una función que a partir de unos datos, cree la tabla de forma dinámica.

El código que vamos a remplazar es:

...

<table>

<thead>

<tr>

<th>Curso</th>

<th>Profesor</th>

</tr>

</thead>

<tbody>

<tr>

<td>Ingeniería de SW</td>

<td>Pablo Picasso</td>

</tr>

<tr>

<td>Futbol I</td>

<td>Juan Piña</td>

</tr>

<tr>

<td>Algoritmos Genéticos</td>

<td>María de la Rosa</td>

</tr>

</tbody>

</table>

...

El código anterior lo cambiamos por:

...

<table class="table">

<thead class="thead-dark">

<tr>

<th>Curso</th>

<th>Profesor</th>

<th>Créditos</th>

</tr>

</thead>

<tbody id="courses">

</tbody>

...

Note que la parte de la etiqueta tbody, la dejamos vacía.

La función en TypeScript que vamos a definir, reemplaza el elemento tbody que está identificada con id= "courses" por una fila (etiqueta tr) en la tabla, por cada elemento en el arreglo dataCourses.

**Definición de las funciones**

Las funciones que van a manejar el DOM del HTML, las vamos a escribir en un archivo denominado main.ts

El contenido de ese archivo puede ser consultado [en este enlace](https://github.com/Uniandes-isis2603/TallerTypescript/blob/master/main.ts).

| **Una explicación sobre cómo se manipula el DOM desde Typescript puede ser encontrada**[**aquí**](https://github.com/Uniandes-isis2603/recursos-isis2603/wiki/dom)**.** |
| --- |

En las siguientes secciones se explicará en detalle el contenido de ese archivo.

**Mostrar los cursos en la tabla**

La función renderCoursesInTable lee los datos de un arreglo de objetos curso y finalmente lo despliega en la tabla de cursos actuales (DOM).

const coursesTbody: HTMLElement = document.getElementById('courses')!; // Nodo tbody que tiene el id="courses"

function renderCoursesInTable(courses: Course[]): void {

courses.forEach(c => {

let trElement = document.createElement("tr");

trElement.innerHTML = `<td>${c.name}</td>

<td>${c.professor}</td>

<td>${c.credits}</td>`;

coursesTbody.appendChild(trElement);

});

}

Varias cosas para resaltar en la función anterior:

* La primera línea define una constante de tipo HTMLElement. document.getElementById('courses') puede retornar un valor null si no encuentra el elemento. Sin embargo, un valor null no puede ser asignado a una variable de tipo HTMLElement, entonces se utiliza el símbolo de exclamación ! para quitar la posibilidad de retornar null o undefined y poder hacer la asignación.
* El argumento de la función renderCoursesInTable define que sólo puede recibir un array de objetos Course y la función no devolverá ningún valor (void).
* La función como primer paso recorre uno a uno el arreglo de los cursos con la función de alto orden forEach.
* forEach recibe una función que es la que va a aplicar sobre cada elemento del arreglo. La expresión (c) => {} es una función flecha o *arrow function* que simplifica la declaración de una función y a su vez realiza un *bind* del objeto this.
* Esta función, por cada curso crea una etiqueta tr y tres etiquetas hijas td con la información del nombre del curso, de los créditos y del profesor.

**Calcular número de créditos actuales**

La función getTotalCredits recorre todos los cursos actuales para obtener el total de créditos del estudiante.

function getTotalCredits(courses: Course[]): number {

let totalCredits: number = 0;

courses.forEach((course) => totalCredits = totalCredits + course.credits);

return totalCredits;

}

**Filtrar por nombre del curso**

Es este apartado vamos a crear un filtro sobre el nombre del curso. El usuario debe agregar el nombre o iniciales del curso y deben aparecer las coincidencias respectivas.

En el HTML agregamos un form con un input y un botón, justo encima de la tabla, así:

<form style="margin:10px;">

<div class="form-row">

<div class="col">

<input type="text" class="form-control" id="search-box" placeholder="Filtrar por nombre">

</div>

<div class="col">

<button type="button" id="button-filterByName" class="btn btn-success">Aplicar</button>

</div>

</div>

</form>

El archivo main.ts contiene dos funciones más: *applyFilterByName* para obtener el texto de búsqueda, limpiar la tabla y llamar a la búsqueda y finalmente *searchCourseByName* que se encarga de ejecutar la búsqueda mediante el uso de [filter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter).

function applyFilterByName() {

let text = inputSearchBox.value;

text = (text == null) ? '' : text;

clearCoursesInTable();

let coursesFiltered: Course[] = searchCourseByName(text, dataCourses);

renderCoursesInTable(coursesFiltered);

}

function searchCourseByName(nameKey: string, courses: Course[]) {

return nameKey === '' ? dataCourses : courses.filter( c =>

c.name.match(nameKey));

}

Finalmente, debemos relacionar el evento onClick con la función *applyFilterByName* de la siguiente forma:

btnfilterByName.onclick = () => applyFilterByName();

**Compilación de los archivos .ts**

Antes de incluir los scripts en el HTML necesitamos compilar los archivos con extensión .ts. Para esto vamos en VSCode al menú Terminal > Run Tasks > tsc: build - tallerTypesript/tsconfig.json. Esto creará una carpeta scripts dentro del proyecto la cual contiene los archivos course.js, dataCourse.js y main.js.

**Incluir los scripts en el Html**

Para que el navegador pueda cargar los archivos JavaScript compilados a partir de los archivos TypeScript debemos incluir las siguientes líneas en el archivo index.html, antes del cierre de la etiqueta <body>:

...

<body>

...

<script type="module" src="./scripts/course.js" ></script>

<script type="module" src="./scripts/dataCourses.js" ></script>

<script type="module" src="./scripts/main.js"></script>

</body>

</html>

**Solución Simple**

La figura muestra una solución para el paso 3 del taller.

| **Imgur** |
| --- |
| Figura 5: Paso 3 |

**https://camo.githubusercontent.com/d415e53fcf416a5f3b7a2f84d470d3ac4afa1a670a8d925a20e4ade513e7f9e6/68747470733a2f2f692e696d6775722e636f6d2f54784d6a5442662e706e67 Ejercicio TypeScript**

* Completar la información personal del estudiante a partir de un objeto de TypeScript creado a partir de una clase Student.
* Realizar un filtro por rangos de créditos de los cursos. Por ejemplo, fijar un valor mínimo de créditos y un valor máximo y al aplicar este filtro deben aparecer los cursos que pertenezcan a este intervalo.