

Laboratorio III

TypeScript (parte 1)

Clase 01

Maximiliano Neiner

Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
- Tipos de datos
- Funciones

Temas a Tratar

- Introducción a TypeScript
 - Inconvenientes con JavaScript
 - TypeScript
- Instalación de TypeScript
- Tipos de datos
- Funciones

JS - ES5

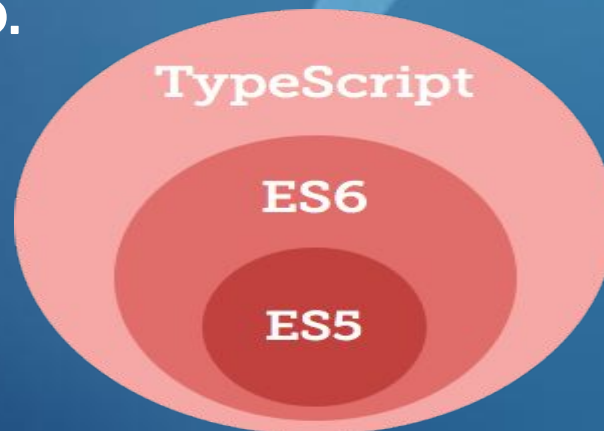
- Falta de tipado fuerte y estático (tipado dinámico).
- El compilador no ayuda.
 - Hay que ejecutar los test (si se tienen).
- La IDE tampoco ayuda.
 - No se puede refactorizar de forma automática.
 - El auto completado es muy limitado.
 - No se puede navegar a la implementación.
- La herencia no es limpia (con prototipos).
- Los patrones de diseño OO no se pueden aplicar directamente.
- Falta de interfaces y módulos.

Temas a Tratar

- Introducción a TypeScript
 - Inconvenientes con JavaScript
 - TypeScript
- Instalación de TypeScript
- Tipos de datos
- Funciones

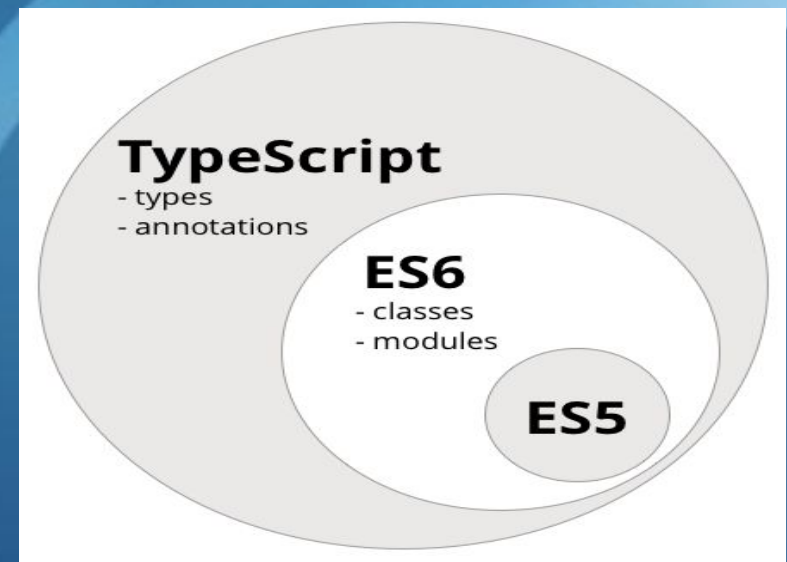
TypeScript (1/2)

- **TypeScript** es un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft, que permite crear aplicaciones Web robustas en JavaScript.
- **TypeScript** no requiere de ningún tipo de plugin, puesto que lo que hace es generar código JavaScript que se ejecuta en cualquier navegador, plataforma o sistema operativo.
- **TypeScript** es un "transpilador", es decir, un compilador que se encarga de traducir las instrucciones de un lenguaje a otro.



TypeScript (2/2)

- Añade tipos estáticos a JavaScript ES6.
 - Inferencias de tipos.
 - Tipos opcionales.
- El compilador genera código JavaScript ES5 (Navegadores actuales).
- Orientado a Objetos con clases. (No como ES5).
- Anotaciones (ES7).



Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
 - Línea de comando
 - IDEs
- Tipos de datos
- Funciones

Instalación de TypeScript (1/3)

- Se necesita la instalación de un servidor *NodeJS*.
- Para descargar *NodeJS* hay que ir a nodejs.org.
 - Una vez instalado, comprobaremos la instalación escribiendo sobre la terminal el comando:

```
node -v
```

- Si indica la versión de *NodeJS*, el siguiente paso es la descarga de **TypeScript**, con el gestor de paquetes *npm*.

```
npm install -g typescript@latest
```

- Se verifica con:

```
tsc -v
```

Instalación de TypeScript (2/3)

- El siguiente paso será crear una carpeta donde trabajar.
 - una vez creada, navegaremos a través de la terminal a la carpeta y escribiremos el siguiente comando:

```
tsc --init
```

- Con este comando se generará el archivo de configuración ***tsconfig.json***, que utilizará TypeScript para compilar la información.
- El archivo creado tendrá un aspecto similar al siguiente:

Instalación de TypeScript

(3/3)

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig to read more about this file */

    /* Projects */
    // "incremental": true,                          /* Save .tsbuildinfo */
    // "composite": true,                             /* Enable composite */
    // "tsBuildInfoFile": "./.tsbuildinfo",           /* Specify the output file for tsbuildinfo */
    // "disableSourceOfProjectReferenceRedirect": true, /* Disable project reference redirects */
    // "disableSolutionSearching": true,              /* Optimize project loading */
    // "disableReferencedProjectLoad": true,          /* Reduce the number of projects loaded */

    /* Language and Environment */
    "target": "es2016",                               /* Set the JavaScript target */
    // "lib": [],                                     /* Specify the library files to be included in the compilation */
    // "strict": true,                                /* Enable strict mode */
  }
}
```

- Este archivo tiene que estar en el directorio raíz del proyecto.

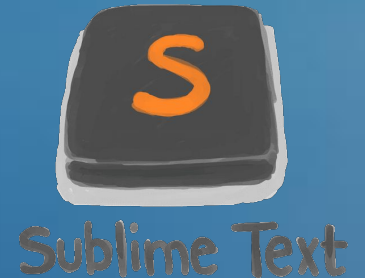
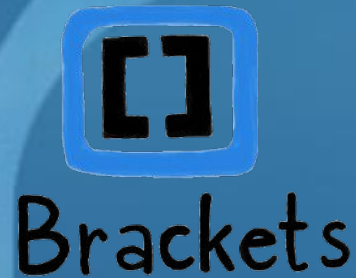
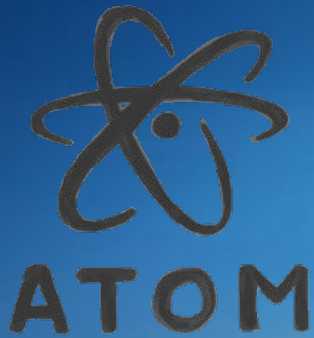
Demo

- **Instalación y comprobación**

Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
 - Línea de comando
 - IDEs
- Tipos de datos
- Funciones

IDE (1/6)



IDE (2/6)

- **Visual Studio Code** es una IDE de Microsoft, pero de código abierto.
- Posee, entre otros:
 - Una terminal integrada.
 - Intellisense.
 - Consola de depuración.
 - Gestor de extensiones.
 - Etc.
- Permite automatizar tareas para el debugging.

Demo

. IDE

IDE (3/6)

- Una vez abierto el Visual Studio Code, lo prepararemos para poder 'debuggear' los distintos archivos de nuestro proyecto.
- Primero, se modificará el archivo **tsconfig.json**.
 - Descomentar el campo **"sourceMap" : true**.
Esto permitirá 'debuggear' desde la consola de cualquier navegador Web y desde la consola integrada del Visual Studio Code.
- Transpilar **todos** los archivos TypeScript del proyecto.
- Abrir el menú de Terminal y seleccionar:
 - **Ejecutar tarea de compilación**
 - Elegir **tsc: compilación tsconfig.json**, para que transpile con las configuraciones elegidas.

IDE (4/6)

- Para que se transpile un archivo TypeScript, se utiliza el siguiente comando, desde la terminal:

```
tsc <nombre_archivo>.ts
```


- Para agregar una inspección sobre un archivo, se escribirá el siguiente comando:

```
tsc -w <nombre_archivo>.ts
```

- Para juntar varios archivos .ts en un solo archivo de salida:

```
tsc --outFile <main> <arch1> <arch2>
```

IDE (5/6)

- Una vez generados los archivos JavaScript, preparamos el Visual Studio Code, para poder 'debuggear' los distintos archivos de nuestro proyecto.
- Abrir el menú de de depuración:
 - Ver -> Ejecutar (Ctrl + Shift + D)
 - O pulsar el ícono de la barra de actividades 
 - Para personalizar 'Ejecutar y depurar' se seleccionará el entorno (Node.js).
 - Opcionalmente, se podrá generar, dentro de la carpeta **.vscode** el archivo **launch.json**, al que se le agrega la configuración de **Iniciar Programa**.

Para personalizar
Ejecutar y depurar
cree un archivo
launch.json.

IDE (6/6)

```
"version": "0.2.0",  
"configurations": [  
  {  
    "name": "Launch Program",  
    "program": "${workspaceFolder}/app.js",  
    "request": "launch",  
    "skipFiles": [  
      "<node_internals>/**"  
    ],  
    "type": "pwa-node"  
  },  
  {  
    "type": "node",  
    "request": "launch",  
    "name": "Iniciar el programa",  
    "skipFiles": [  
      "<node_internals>/**"  
    ],  
    "program": "${file}"  
  }  
]
```

Agregar configuración...

- Reemplazar *app.js* por el archivo a ser depurado.
- Iniciar la depuración con F5.

Demo

- IDE configuración

Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
- Tipos de datos
- Funciones

Tipos de datos

- Tipado estático o fuertemente tipado:
 - Se debe de definir el tipo de dato, obligando a que no pueda haber errores con los tipos de datos.
- Tipado dinámico o débilmente tipado:
 - No se deben de o tiene porque especificar el tipo de dato (PHP, JavaScript).

```
var a = 3;  
var b = "hola";  
var c = a + b; // -> resultado 3hola  
if("0" == 0) // -> true  
if("3" === 3) // -> false
```

Tipos en TypeScript

- Boolean
- Number
- String
- Any
- Void
- Array
- Null
- Undefined
- Tuple
- Enum

Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
- Tipos de datos
 - Primitivos
 - Arrays
 - Enums
 - Let vs. Var
- Funciones

Tipos Primitivos (1/3)

- Boolean
 - true o false.

```
var esVerdad : boolean = false;
```

- Number
 - Valores numéricos (enteros, decimales, octales y hexa).

```
var numero : number = 33.78;
```


- Null
 - Cuando un objeto o variable no esta accesible.

```
var obj : object | null = null;
```

Tipos Primitivos (2/3)

- Undefined
 - Es cuando un objeto o variable existe pero no tiene un valor.
- Any
 - Puede ser cualquier tipo de objeto de JavaScript.

```
var cosa : any = "rojo";  
cosa = 3;  
cosa = false;
```



- Void
 - Generalmente usado en funciones.

```
function Avisar() : void { console.log("Cuidado!!!"); }
```

Tipos Primitivos (3/3)

- String
 - Cadenas de caracteres y/o textos.

```
var color : string = "rojo"; //comillas dobles  
color = 'azul';           //comillas simples  
color = `verde`;          //tilde invertido
```

- Plantillas de string
 - Se escriben entre ` (tildes invertidos) y la sintaxis sería:

```
var mensaje : string = 'hola mundo'; //simples o dobles  
var html : string = `

# ${mensaje}</h1>`; //tilde invertido


```

Demo

- Tipos primitivos

Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
- Tipos de datos
 - Primitivos
 - Arrays
 - Enums
 - Let vs. Var
- Funciones

Arrays

- Array
 - Si no se les especifica tipo son ANY.

```
var lista = [1, true, "rojo"];
```

- Con esta sintaxis se puede especificar tipo de elementos.

```
var lista : number[] = [1, 2, 3];
```

```
var lista : Array<number> = [1, 2, 3];
```

```
var lista : number = [1, 2, 3];
```



Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
- Tipos de datos
 - Primitivos
 - Arrays
 - Enums
 - Let vs. Var
- Funciones

Enum

- Los enumerados en TypeScript solo almacenan números para identificar a las constantes.
 - Sin asignación de valores:

```
enum Color { Rojo, Verde, Azul };
```

```
var c : Color = Color.Verde; // 2
```

```
enum Color { Rojo = 2, Verde = 5, Azul = 8 };
```

```
var c : Color = Color.Verde; // 5
```

Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
- Tipos de datos
 - Primitivos
 - Arrays
 - Enums
 - Let vs. Var
- Funciones

LET

- En TypeScript hay dos formas de declarar variables ***var*** y ***let***:
 - ***var*** no tiene un ámbito de bloque (es global), mientras que ***let*** sí.
- ***var***

```
var foo : number = 123;  
if(true) { var foo : number = 456; }  
console.log(foo); // 456
```

- ***let***

```
let foo : number = 123;  
if(true) { let foo : number = 456; }  
console.log(foo); // 123
```

Demo

- Otros tipos

Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
- Tipos de datos
- Funciones
 - Tradicionales
 - Fat Arrow

Funciones (1/4)

- Sintaxis de una función básica:

```
function Identificador([args : tipo]) : TipoRetorno  
{ [return;] }
```

- Con parámetros opcionales

```
function Identificador(param? : tipo) : TipoRetorno  
{ [return;] }
```

- Si no se recibe valor para el parámetro (?), se asigna undefined.

Funciones (2/4)

- Con parámetros predeterminados

```
function Identificador(param : tipo = valor) : TipoRetorno  
{ [return;] }
```

- Si no se recibe valor para el parámetro, se asigna **valor**.

- Con parámetros REST

```
function Identificador(...params : tipo[]) : TipoRetorno  
{ [return;] }
```

- Permiten pasar un array de parámetros.

Funciones (3/4)

- Como variables

```
let saludar : Function = function() : string
{
    return "Hola Mundo!!!";
}
```

```
console.log(saludar()); //Hola Mundo!!!
```

```
function Cuadrado(a:number) : number
{ return a * a; }
```

```
let pot : Function = Cuadrado;
console.log(pot(2)); //4
```


Funciones (4/4)

- Sobrecargas

```
function a(x : number) : void;  
function a(x : string) : void;  
function a(x : boolean) : void;  
function a(x : Array<number>) : void;  
function a(x : any) : void  
    {  
        //implementación  
    }
```

- Este tipo de sobrecarga no tiene mucho sentido porque sería más simple poner un parámetro de tipo *any*.

Demo

- **Funciones**

Temas a Tratar

- Introducción a TypeScript
- Instalación de TypeScript
- Tipos de datos
- Funciones
 - Tradicionales
 - Fat Arrow (función flecha)

Fat Arrow

- El nombre de ‘fat arrow’ => surge como oposición a las ‘flechas finas’ ->.
- Son funciones cuyo propósito es:
 - Omitir las palabras ***function*** y ***return***.
 - Implementar el ***this*** léxico.
- Parámetros con Fat Arrow:

`() => { }` // sin parámetros, lleva paréntesis.

`x => { }` // un parámetro, puede no llevar paréntesis.

`(x,y) => { }` // varios parámetros, lleva paréntesis.

- Parámetros con funciones ‘normales’:

```
function() { .... }
```

```
function(x,y) { .... }
```

Fat Arrow

- Implementar el cuerpo:

```
x => { return x * x; } // bloque
```

```
x => x * x; // expresión, equivalente al anterior.
```

- El bloque de instrucciones se comporta como un cuerpo de función normal.
- Con un cuerpo de expresión, la expresión siempre se devuelve implícitamente. Se puede omitir el ***return***.
- Tener un cuerpo de bloque agregado a un un cuerpo de expresión significa que la expresión retornada es un objeto literal. Se debe poner entre paréntesis.

```
let res = () => { ("nombre" : "juan", "edad" : 23) };
```

Fat Arrow

- Hasta las funciones flecha, cada nueva función define su propio valor de *this*.
 - Un nuevo objeto en el caso de un constructor
 - indefinido en llamadas de función de modo estricto
 - etc.
- Esto resulta ser muy molesto con un estilo orientado a objetos de programación.

Demo

- **Funciones Flecha**



Ejercitación