

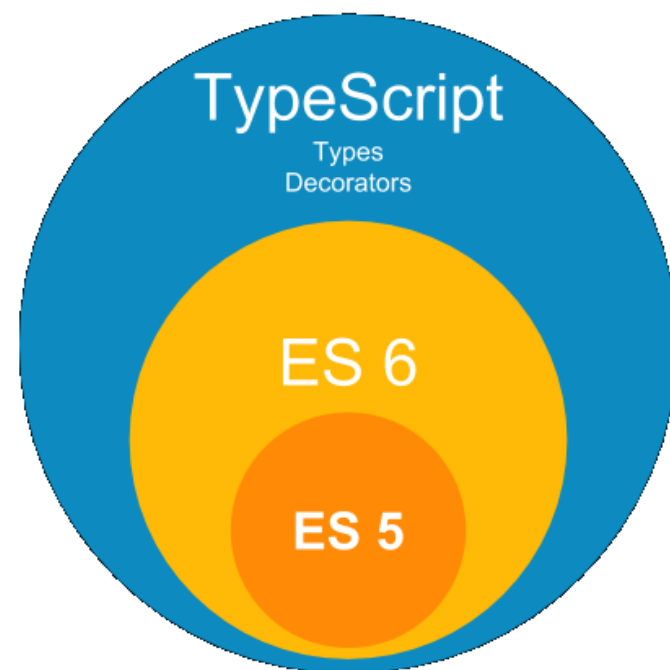


## **Introducción Typescript**



## Typescript

- Lenguaje Libre
- Código abierto
- Creado por Microsoft
- Superconjunto de JavaScript
- Agrega:
  - Tipado estático
  - Clases y objetos





## Typescript

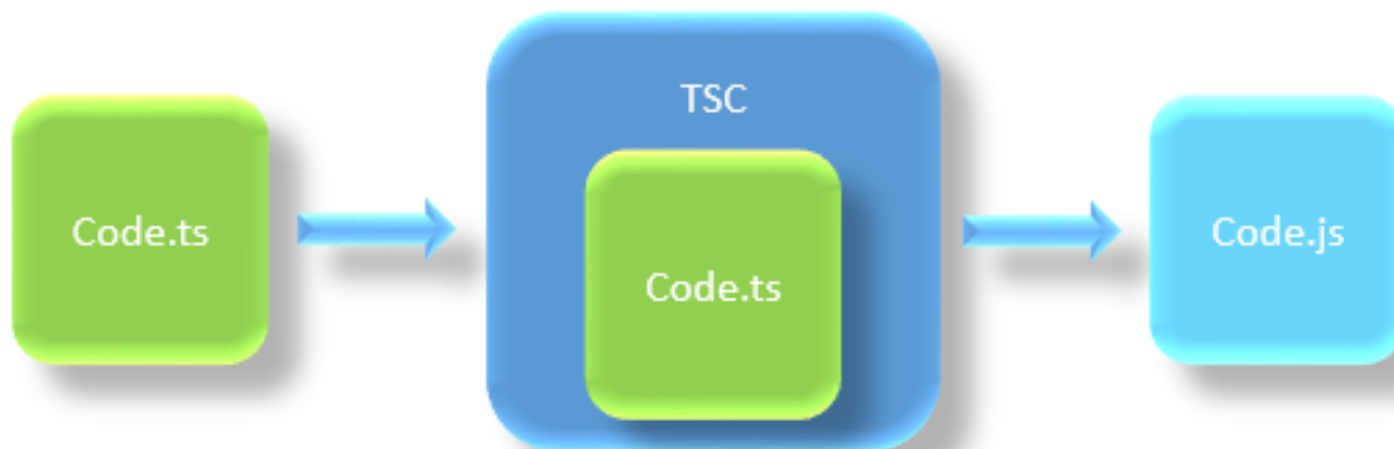
¿Pero el browser ejecuta Typescript?





## Compilador de Typescript

- Convierte el código TS a código JS.
- Empaqueta todos los archivos JS en uno.
- Minifica el archivo JS generado.
- Lo deja listo para incluir en index.html





## Variables

```
let m:number;  
let s:string;  
let b:boolean = false;  
let o:object; // no primitivo  
  
let list: number[] = [1, 2, 3];  
let list: Array<number> = [1, 2, 3];  
  
let notSure: any = 4;  
  
let list: any[] = [1, true, "free"];
```



## Conversiones

```
let i:number;  
let s:string;
```

```
i = 9;  
s = i.toString(); // numero a string
```

```
i = +s; // string a numero  
i = parseInt(s); // string a numero  
i = parseFloat(s); // string a numero
```



## Arrays

```
let lista: Array<number>;  
lista = [];  
lista = new Array<number>();  
  
lista.push(2);  
lista.push(3);  
lista.push(5);  
  
for(let i in lista){  
    console.log(lista[i]);  
}  
  
console.log("size:"+lista.length);
```



## Diccionario

```
let perros:Map<string, Dog>;  
perros = new Map<string, Dog>(); // ES6
```

```
perros["bobby"] = new Dog();  
perros["firulais"] = new Dog();
```

```
for(let k in perros) {  
    console.log("Clave:"+k);  
    console.log(perros[k]);  
}
```





## Funciones

**JS:**

```
function add(x, y) {  
    return x + y;  
}
```

**TS:**

```
function add(x: number, y: number): number {  
    return x + y;  
}
```



## Declaraciones var

```
function f(shouldInitialize: boolean) {  
    if (shouldInitialize) {  
        var x = 10;  
    }  
  
    return x;  
}
```

```
f(true);    // returns '10'  
f(false);  // returns 'undefined'
```



## Declaraciones **let**

```
function f(shouldInitialize: boolean) {  
    if (shouldInitialize) {  
        let x = 10;  
    }  
  
    return x;  
}
```

ERROR DE COMPILACIÓN: "Cannot find name 'x'."



## Declaraciones var

```
function f(x:number) {  
    var x:number = 100;  
}
```

Declaración exitosa



## Declaraciones **let**

```
function f(x:number) {  
    let x:number = 100;  
}
```

**ERROR DE COMPILACIÓN:** "Duplicate identifier 'x'."



## Declaraciones var

```
function f() {  
    var a = 10;  
    return function g() {  
        var b = a + 1;  
        return b;  
    }  
}
```

```
var g = f();  
g(); // returns '11'
```

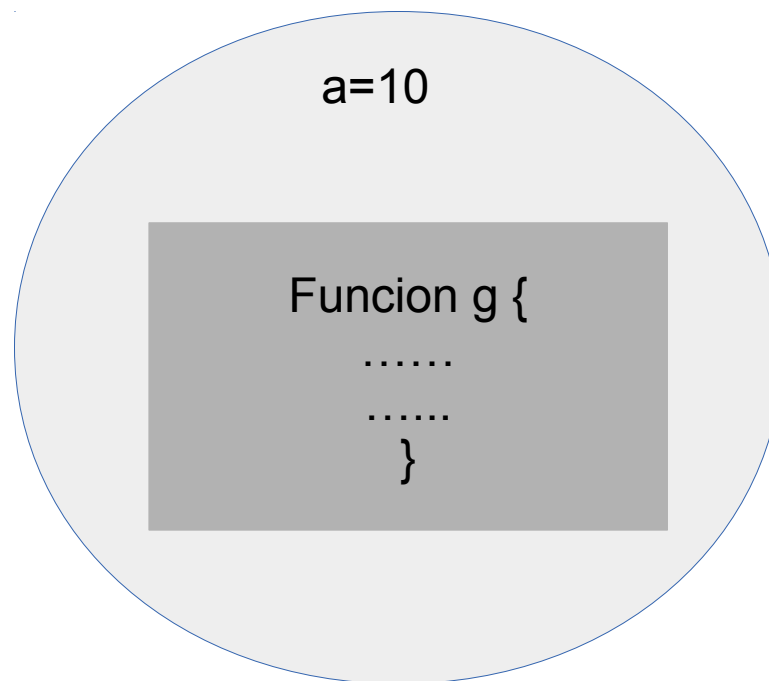
Cada vez que ejecutamos `g()`, se utilizará el valor de “a” declarado en la función `f()`, aunque la misma ya haya terminado su ejecución.



## Declaraciones var

```
function f() {  
  var a = 10;  
  return function g() {  
    var b = a + 1;  
    return b;  
  }  
}
```

```
var g = f();  
g(); // returns '11'
```

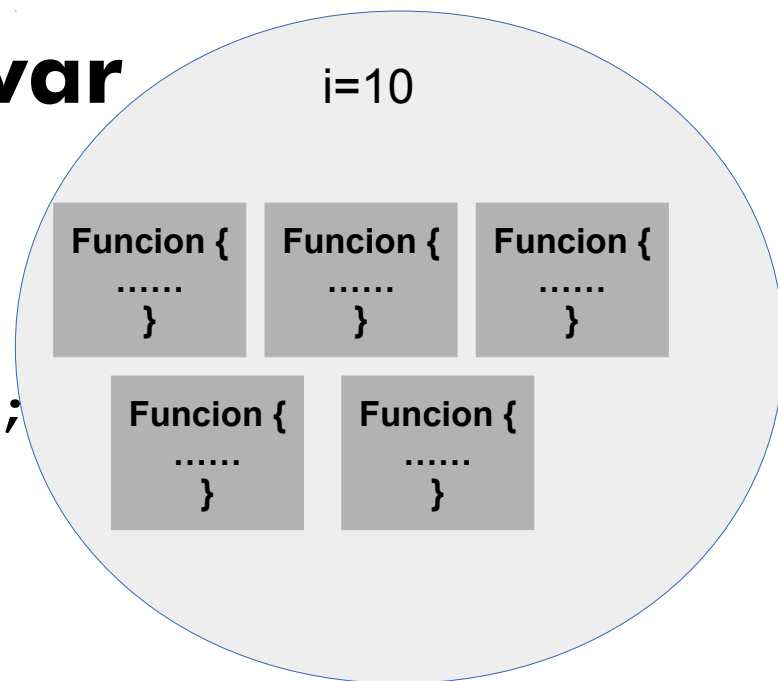


**Closure** es la combinación de una función y el ámbito léxico en donde se declaró dicha función.



## Declaraciones var

```
for (var i = 0; i < 10; i++) {  
    setTimeout(function() {  
        console.log(i);  
    },  
    100 * i);  
}
```



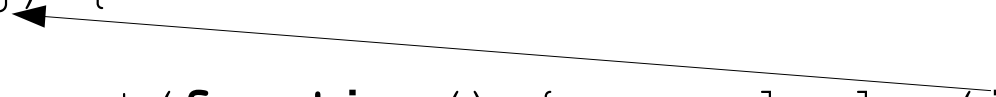
```
10  
10  
10  
10  
10  
10  
10  
10  
10  
10  
10
```





## Declaraciones var

```
for (var i = 0; i < 10; i++) {  
    // capture the current state of 'i'  
    // by invoking a function with its current value  
  
    (function(j) {  
        setTimeout(function() { console.log(j); }, 100 * j);  
  
    })(i);  
}
```



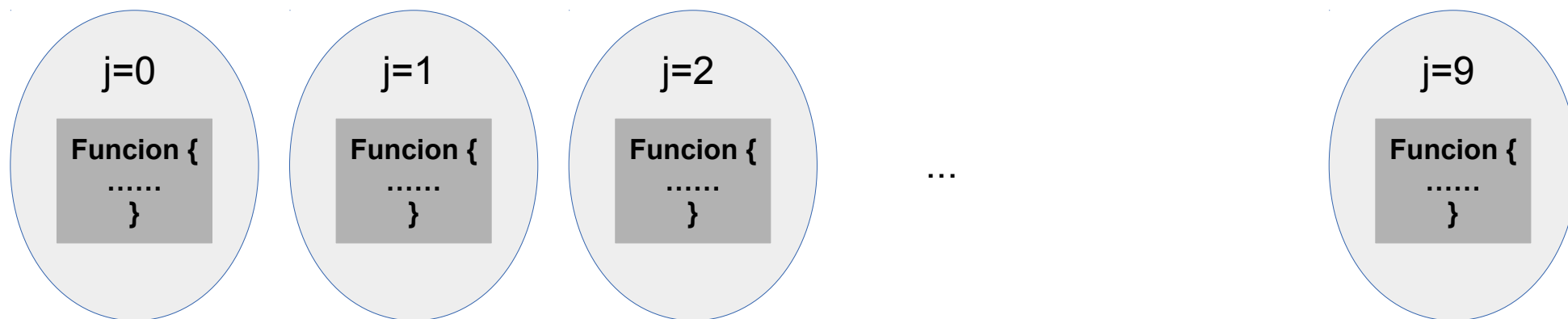
## IIFE: Immediately Invoked Function Expression

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```



## Declaraciones var

```
for (var i = 0; i < 10; i++) {  
    // capture the current state of 'i'  
    // by invoking a function with its current value  
  
    (function(j) {  
        setTimeout(function() { console.log(j); }, 100 * j);  
  
    })(i);  
}
```





## Declaraciones **let**

```
for (let i = 0; i < 10; i++) {  
  
    setTimeout(function() {  
        console.log(i);  
    },  
    100 * i);  
}
```

Al usar **let** en un loop, se crea un nuevo scope por iteración.

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```



## Funciones como argumentos

```
function pot(x:number) :number {  
    return x*x;  
}
```

```
function calcular(n:number, fun:any) :number {  
    return fun(n);  
}
```

```
console.log(calcular(2,pot)); // Returns '4'
```



## Funciones como argumentos

```
function pot(x:number):number {  
    return x*x;  
}
```

```
function suma(x:number, y:number):number {  
    return x+y;  
}
```

```
function calcular(n:number, fun:  
    (x:number)=>number):number  
{  
    return fun(n);  
}
```

```
console.log(calcular(2, suma)); // Error compilación
```



## Clases

```
class Main
{
    constructor()
    {
        console.log("constructor");
    }
}
```

```
let m = new Main();
```

- 1 solo constructor



## Clases

```
class Greeter {  
  
    greeting: string;  
  
    constructor(message: string) {  
        this.greeting = message;  
    }  
  
    greet(): string {  
        return "Hello, " + this.greeting;  
    }  
}  
  
let greeter:Greeter = new Greeter("world");  
greeter.greet(); // Hello world
```



## Métodos

```
class Main
{
    constructor()
    {
        console.log("constructor");
    }

    miMetodo(a:number,b:string):number
    {
        console.log(b);
        return a+1;
    }
}

let m:Main = new Main();
let r:number = m.miMetodo(2,"mensaje");
console.log(r);
```





## Herencia

```
class Animal {  
    move(dist: number = 0) {  
        console.log(`Animal moved ${dist}.`);  
    }  
}
```

```
class Dog extends Animal {  
    bark() {  
        console.log('Woof! Woof!');  
    }  
}
```

```
let dog: Dog = new Dog();  
dog.bark();  
dog.move(10);  
dog.bark();
```

- Herencia simple
- Múltiples interfaces



```
class Person {  
    firstName: string;  
    lastName: string;
```

**Super**

```
    constructor (fName: string, lName: string) {  
        this.firstName = fName;  
        this.lastName = lName;  
    }  
  
class Employee extends Person {  
    empID: string;  
    designation: string;  
  
    constructor (fName: string, lName: string,  
                eID: string, desig: string) {  
        super(fName, lName);  
        this.empID = eID;  
        this.designation = desig;  
    }  
}
```



## Métodos desde dentro de la clase

```
class Animal {  
    move(dist: number = 0) {  
        console.log(`Animal moved ${dist}.`);  
    }  
}
```

```
class Dog extends Animal {  
    bark() {  
        console.log('Woof! Woof!');  
    }  
    beDog() {  
        super.move(10);  
        this.bark();  
    }  
}
```



## Modificadores de acceso

- **Public, Private y Protected**
- Por default atributos y métodos son públicos.
- “private” solo se accede desde la clase.
- “protected” es como private pero las clases que heredan tienen acceso.



## Getters y Setters

```
class Employee {  
  
    private _fullName: string;  
  
    get fullName(): string {  
        return this._fullName;  
    }  
  
    set fullName(newName: string) {  
        this._fullName = newName;  
    }  
}  
  
let e:Employee = new Employee();  
e.fullName = "Juan";
```



## Atributos y métodos estáticos

```
class Employee {  
  
    static saludo: string = "HOLA";  
  
    static metodoEstatico():void {  
        console.log("hola");  
    }  
}  
  
console.log(Employee.saludo);  
  
Employee.metodoEstatico();
```



## Bibliografía

- `https://code.visualstudio.com/docs/typescript/typescript-tutorial`
- `https://developer.mozilla.org/es/docs/Web/JavaScript/Closures`
- `https://www.typescriptlang.org/docs/handbook/basic-types.html`