



DECODED

ACADEMY

FULL STACK DEVELOPER BOOTCAMP

Desarrollamos
{ talento }

FUNDAMENTOS DE PROGRAMACIÓN

INTRODUCCIÓN A TYPESCRIPT

The background of the slide features a light gray polygonal grid pattern. Overlaid on this are several large, semi-transparent orange triangles. One triangle is positioned in the upper left quadrant, pointing downwards. Another is at the bottom left, pointing upwards. A smaller orange diamond shape is located near the top center. A single horizontal orange line segment is located in the middle right area of the slide.

Historia

TypeScript es un lenguaje de programación libre y de código abierto creado por [Anders Hejlsberg](#), creador y diseñador de c#, lo que realizamos con typescript es expandir la funcionalidad de Javascript

Instalación

Para instalar Typescript ejecutamos los siguientes comandos.

```
$ npm i -g typescript
```

Para gestionar un servidor que funcione como Live-server

```
$ npm i -g live-server
```

Preparación del proyecto.

Crearemos un archivo vacío con el siguiente nombre **tsconfig.json**

```
{  
  "compilerOptions": {  
    "target": "es6",  
    "sourceMap": true  
  }  
}
```

Esto nos generará los archivos .js a partir de .ts

Preparación del proyecto

Creamos un archivo index.html

donde cargaremos el script src='app.js'

Preparación del proyecto.

Levantamos nuestro servidor.

```
$ live-server
```

Preparacion del proyecto

Para utilizar typescript debemos transpilar nuestro codigo mediante el comando

```
$ tsc [nombre-del-archivo]
```

Sintaxis



Tipos

Los tipos existentes en typescript son los mismo que comparten otros lenguajes como javascript etc.

- Booleans
- Strings
- Number
- Array
- Enum
- Tuple
- Any
- Void
- Never
- null/Udefined.

Tipos

Podemos ver tipos un poco especiales.

- Any -> Recibe cualquier tipo de dato.
- Void -> Vacio se utiliza en funciones que no retornan nada.
- Tuple -> Array asociativo definido bidimensional.
- Never -> Valores que se suponen que nunca ocurren./li>

Variables y constantes.

A diferencia de Javascript aquí las variables y las constantes vienen tipadas con el tipo que se quiere utilizar.

```
let name: string = "Xavi";
let apellido: string = "Rodriguez";
let readonly miConstante: string = "Constante";
const miConstante: string = 'Constante'; // ES6
```

Arrays

```
let names: string[] = ['Javier', 'Emma', 'John', 'Sophia', 'Emma'];  
// ○  
let names: Array<string> = ['Javier', 'Emma', 'John', 'Sophia', 'Emma'];
```

Funciones

A diferencia de Javascript aquí las variables y las constantes vienen tipadas con el tipo que se quiere utilizar.

```
function sum(x: number, y:number ):number {  
    return x + y;  
}  
let convertToString (x: any): string => String(x)
```

Clases

Las clases se definen al igual que JavaScript simplemente que se definen los tipos de los atributos de la clase, se parece a otros lenguajes de programación



```
class Persona{  
    nombre: string;  
    apellidos: string;  
    private edad: number;  
    constructor(nombre:string,apellidos:string,edad:number) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.edad = edad  
    }  
    get Nombre():string{  
        return this.nombre;  
    }  
    set Nombre(nombre:string) {  
        this.nombre = nombre;  
    }  
}  
let persona = new Persona("xavi","apellidos",23);  
console.log(persona.Nombre);  
persona.Nombre ="Pepe";  
console.log(persona.Nombre);
```


Clases

Como las clases de javascript también podemos heredar y ampliar sus propiedades.

Pero nos permite definir el ambito de aplicación de las funciones y los atributos de la clase

- Public
- Private
- Protected

Clases



Por ejemplo podemos ampliar la funcionalidad de nuestras clases y proteger la clase padre

```
class Persona {  
    protected nombre: string;  
    protected constructor(nombre: string) {  
        this.nombre = nombre;  
    }  
}  
class PersonaMayor extends Persona {  
    private apellido: string;  
    constructor(nombre: string, apellido: string) {  
        super(nombre);  
        this.apellido = apellido;  
    }  
    public datos() {  
        console.log(this.apellido + ' ' + this.nombre);  
    }  
}  
let personaMayor = new PersonaMayor('xavi', 'rodriguez');  
personaMayor.datos();  
let persona = new Persona('Xavi'); //Error
```

Clases

También nos permite definir elementos staticos tanto para los parametros de forma staticas

definiendo la palabra reservada **static** delante del atributo.

Clases



Podemos utilizar también clases de tipo abstractas, estas clases tienen la particularidad que no es necesario implementar la clases

```
abstract class Persona {  
    nombre: string;  
    constructor(public name: string) {  
        this.nombre = name;  
    }  
    abstract dameNombre(): void;  
}  
  
class personita extends Persona {  
    constructor() {  
        super('Xavi');  
    }  
    dameNombre(): void {  
        console.log(this.nombre);  
    }  
}  
let yo = new personita();  
yo.dameNombre();
```

Interfaces

Las interfaces nos permiten crear contratos que deben firmar para poder utilizarlos, y además en una clase podemos implementar distintas interfaces.

```
interface iUser {  
    name: string;  
    getNombre(): string;  
}  
class User implements iUser {  
    name: string;  
    constructor(nombre: string) {  
        this.name = nombre;  
    }  
    getNombre(): string {  
        return this.name;  
    }  
}
```

```
interface User {  
    firstName: string;  
    lastName: string;  
}  
  
function printUserInfo(user: User) {  
    console.log(user.firstName);  
}  
  
let myObj: User = { firstName: 'John', lastName: 'Doe' };  
printUserInfo(myObj);
```

```
interface User {  
    firstName: string;  
    lastName: string;  
    address: {  
        city: string;  
        zipcode: number;  
        street: string;  
    };  
}  
  
let myObj: User = {  
    firstName: 'John',  
    lastName: 'Doe',  
    address: {  
        city: 'NY',  
        zipcode: 23001,  
        street: '5 Av',  
    },  
};
```

Propiedades optionales

```
interface User {  
    firstName: string;  
    lastName: string;  
    email?: string;  
}  
let myObj0: User = { firstName: 'John', lastName: 'Doe' }; // ok  
  
let myObj1: User = { firstName: 'John', lastName: 'Doe', email: '44234'  
}; // ok  
  
let myObj2: User = { firstName: 'John', lastName: 'Doe', cif: '44234' };  
// Error
```

Propiedades dinámicas

```
interface User {  
    firstName: string;  
    lastName: string;  
    [key: string]: any;  
}  
let myObj0: User = { firstName: 'John', lastName: 'Doe' }; // ok  
  
let myObj1: User = { firstName: 'John' }; // Error no required lastName  
key  
  
let myObj2: User = { firstName: 'John', lastName: 'Doe', cif: '44234' };  
// ok  
  
let myObj3: User = { firstName: 'John', lastName: 'Doe', edad: 12 }; //  
ok
```

Union types

```
interface User {  
    name: string;  
    id: string | number;  
}  
let myObj0: User = { name: 'John', id: '4432R' } // ok  
  
let myObj1: User = { name: 'John', id: 1283 } // ok
```

```
interface User {  
    name: string;  
    hair: 'castaño' | 'rubio';  
}  
let myObj0: User = { name: 'John', hair: 'castaño' }; // ok  
  
let myObj1: User = { name: 'John', hair: 'rubio' }; // ok  
  
let myObj2: User = { name: 'John', hair: 'moreno' }; // Error
```

```
interface Name {  
    name: string;  
    surname: string;  
    nickname?: string;  
    dni: string | number;  
}  
  
interface User {  
    fullname: Name;  
    hair: 'castaño' | 'rubio';  
    edad: 18 | '18';  
}  
  
let juan: User = {  
    fullname: {  
        name: 'Juan',  
        surname: 'Perez',  
        nickname: 'Garn',  
    },  
    hair: 'castaño',  
    edad: 18,  
    dni: '33556677R',  
};
```

Type Aliases

```
type Hair = 'castaño' | 'rubio';

interface User {
  name: string;
  id: Hair;
}
let myObj0: User = { name: 'John', hair: 'castaño' }; // ok
let myObj1: User = { name: 'John', hair: 'rubio' }; // ok
let myObj2: User = { name: 'John', hair: 'moreno' }; // Error
```

```
interface UserLogged {  
    email: string;  
    name: string;  
    id: number;  
}  
interface UserAnonymous {  
    id: number;  
}  
  
type User = UserLogged | UserAnonymous;  
  
let myObj0: User = { name: 'John', email: 'john@doe.com', id: 1233 }; // ok  
  
let myObj1: User = { id: 1234321 }; // ok
```

```
type StringOrNumber = string | number;
type Text = string | { text: string };
type NameLookup = Dictionary<string, Person>;
type ObjectStatics = typeof Object;
type Callback<T> = (data: T) => void;
type Pair<T> = [T, T];
type Coordinates = Pair<number>;
type Tree<T> = T | { left: Tree<T>; right: Tree<T> };
```

Módulos

Podemos utilizar los módulos para exportar e importar secciones de código para ello utilizamos la especificación de ecmascript6

```
// user.interface.ts
export interface iUser {
  name: string;
}

// app.ts
import { iUser } from './user.interface.ts';
```

NameSpaces

Los namespaces nos permite extender los módulos aglutinando diversos módulos en un mismo namespace de tal manera que es accesible a partir del mismo

```
namespace figuras {  
    export class Rectangulo {  
        prueba() {  
            console.log('Rectangulo');  
        }  
    }  
}
```

NameSpaces

```
namespace figuras {  
    export class Circulo {  
        prueba() {  
            console.log('Circulo');  
        }  
    }  
}  
// Para llamar por ejemplo circulo.  
let circulo = new figuras.Circulo.prueba();
```