*Clase 03. TYPESCRIPT*

NODEJS

**NODE & NPM**

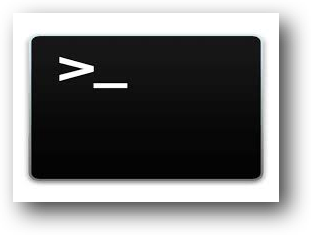
**¿QUÉ APORTA NodeJS?**

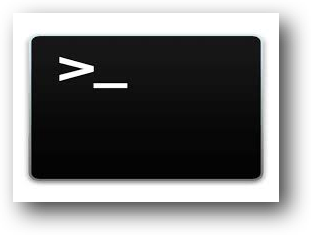
Es considerada una tecnología de servidor que hace aportes a un proyecto javascript aunque sea con tecnología Front-End.  
Repasemos, ¿por qué NodeJS?  
  
Uno de los requisitos que tenemos a la hora de configurar nuestras herramientas para el trabajo con Angular es la instalación del Runtime de NodeJS 🤔 Esto no significa que usaremos NodeJs, pero sí utilizaremos una herramienta que viene con su instalación: npm.  
  
ANGULAR-CLI utiliza muchas configuraciones que dependen internamente de node y npm que se emplean en el entorno de desarrollo y no se trasladan al proyecto compilado 🚀.

Un proyecto ANGULAR finalizado no se encuentra limitado a la plataforma de node para funcionar, sino que puede hacerlo con cualquier servidor web que soporte html, css y javascript.

**PROYECTOS CON NODE**

* Configuración
* Codificación
* Ejecución entorno de desarrollo
* Pruebas / Test
* Preparación de publicación
* Creación del proyecto

**CREACIÓN DEL PROYECTO**  
Consiste en la creación de un archivo package.json. que es un json y que se puede generar manualmente, pero npm nos ofrece la posibilidad hacerlo mediante los siguientes comandos:  
  
npm init  
Nos va preguntando por valores que se irán cargando en el archivo package.json

npm init --yes  
Responde que sí a todas las preguntas que nos va haciendo la generación del packaga.json

**CONFIGURACIÓN**

Es el manifiesto de nuestro proyecto. Además, administra los paquetes y maneja la publicación del proyecto.

**CODIFICACIÓN**  
Con saber programar en javascript, solo vamos a necesitar crear un archivo con extensión .js

console.log("--- Aplicacion de consola ---");

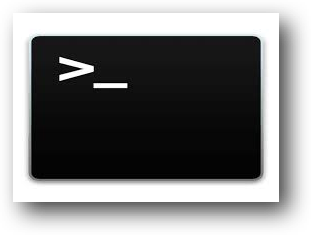
console.log('Nuestra primera aplicacion puro javascript');

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

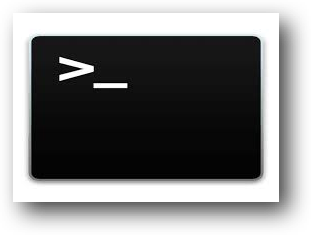
**EJECUCIÓN EN DESARROLLO**

Mientras estamos en tiempo de desarrollo necesitamos ir probando nuestro código

Comando de Node

$ node index.js

Configuración del package.json

$ npm start

**PRUEBAS / TEST**

Las pruebas no forman parte de las funcionalidades nativas del entorno de node. Mientras que las alternativas de test son:

* Generación nativa de pruebas (poco utilizada).
* Mediante la instalación de librerías externas.

**PREPARACIÓN DE PUBLICACIÓN**

Javascript es interpretado según el entorno (browser o runtime), por esto no requiere de un proceso de compilación. Sin embargo, es muy común atravesar un proceso de empaquetado o conversión a código estándar EcmaScript compatible.

Este proceso -en general- se lleva adelante mediante librerías especializadas debidamente configuradas para esta tarea.

TYPESCRIPT

**¿QUÉ ES TYPESCRIPT?**

TypeScript es un lenguaje de programación de código abierto creado por Microsoft en el año 2012 que implementa mecanismos de programación orientados a objetos.



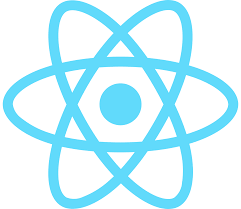
Se trata de un superset de JavaScript que extiende su sintaxis.

Al compilar, se genera código JavaScript ya que el navegador no puede interpretar TypeScript. Este proceso se conoce como transpilar que significa generar un código en un lenguaje específico a partir de otro.

**VENTAJAS**

* Extiende a Superset de Javascript.
* Ofrece tipado estricto y flexible.
* Mejora la legibilidad del código.
* Permite usar nuevas características.

**USOS DE TYPESCRIPT**

Es muy común encontrar typescript del lado del FrontEnd

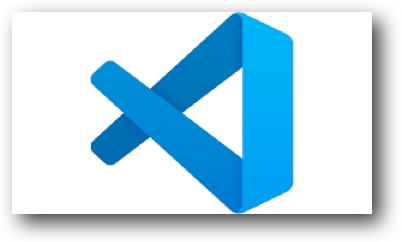


Pero también en el backend FrontEnd



**INSTALAR Y CONFIGURAR**

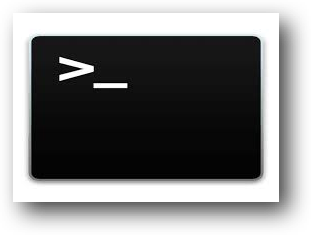
Vamos a necesitar tener instalado nodejs y un editor de código:



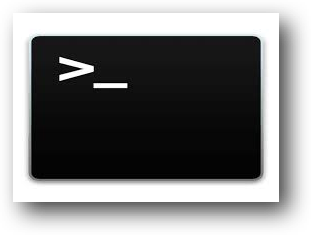
El editor Visual Studio Code viene configurado para aprovechar al máximo TypeScript.

**PROBANDO TYPESCRIPT**

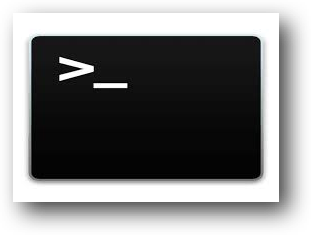
Observemos un ejemplo de flujo de trabajo con typescript

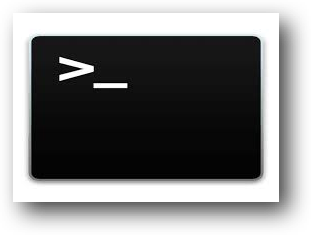
npm install -g typescript

Podemos crear un folder/carpeta y dentro un archivo .ts

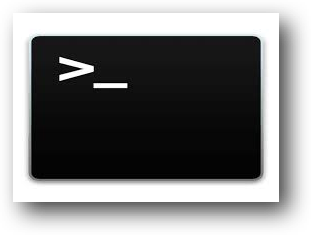
Touch index.ts

Dentro del index.ts se puede escribir un código typescript válido

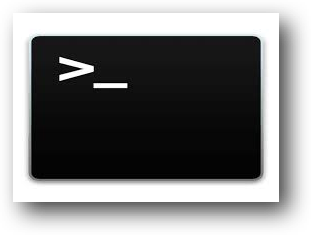
npx tsc index.ts --outFile dist/index.js

tsc index.ts --outFile dist/index.js

**COMPILACIÓN**

npx tsc index.ts

El proceso de compilación parte de un archivo index.ts al cual se le ejecuta el comando de compilación ts y obtenemos un archivo index.js. Esto es necesario para que el código pueda ser interpretado mediante un browser o por el runtime de node.

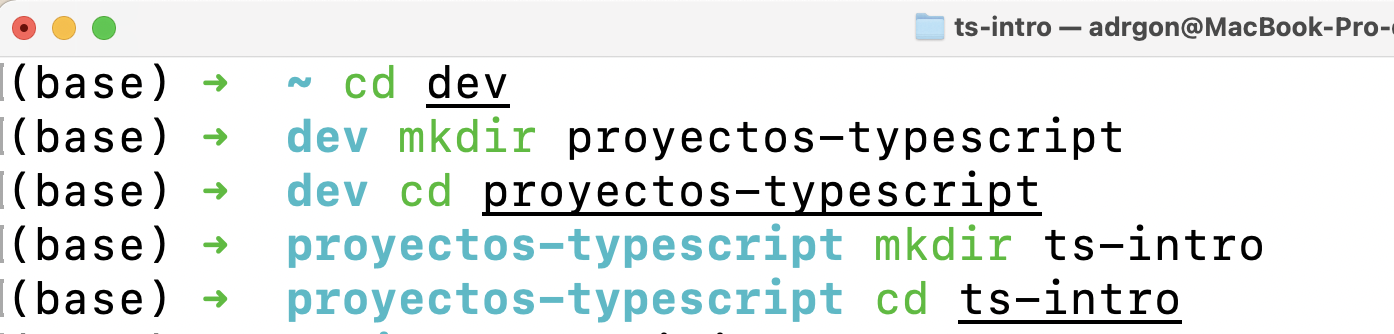
npx tsc --watch index.ts

Sirve para observar los cambios que se producen en el archivo index.ts

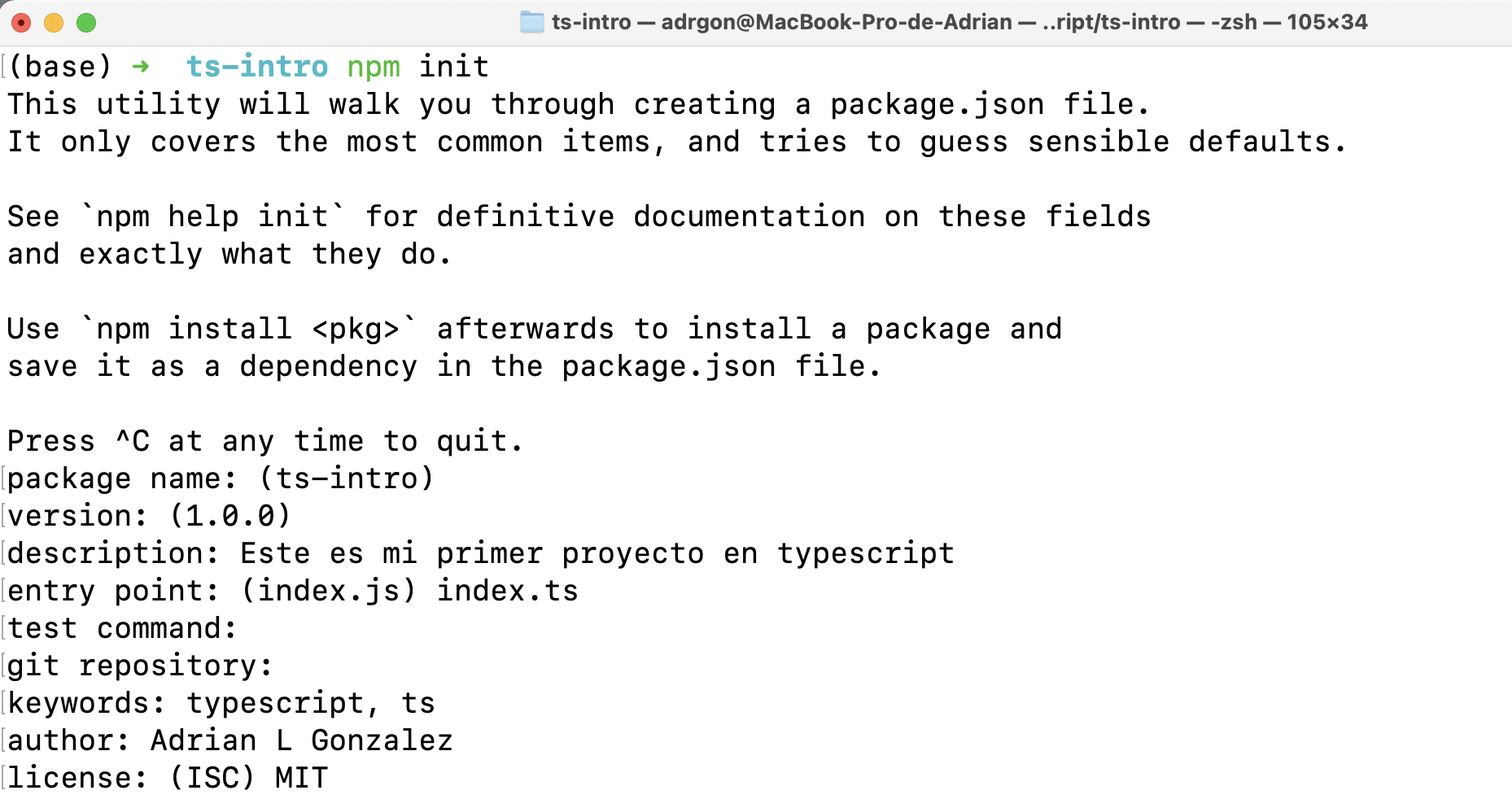
Creamos un proyecto con typescript

**ORGANIZACIÓN DE ARCHIVOS**

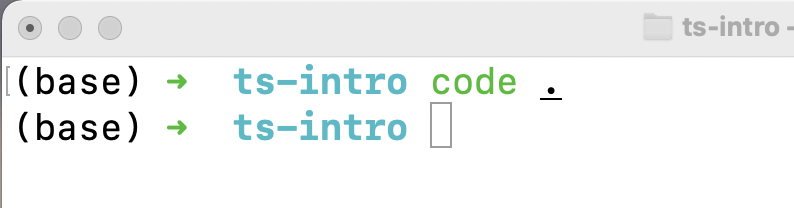
Organizamos archivos y carpetas



**GENERACIÓN PACKAGE.JSON**

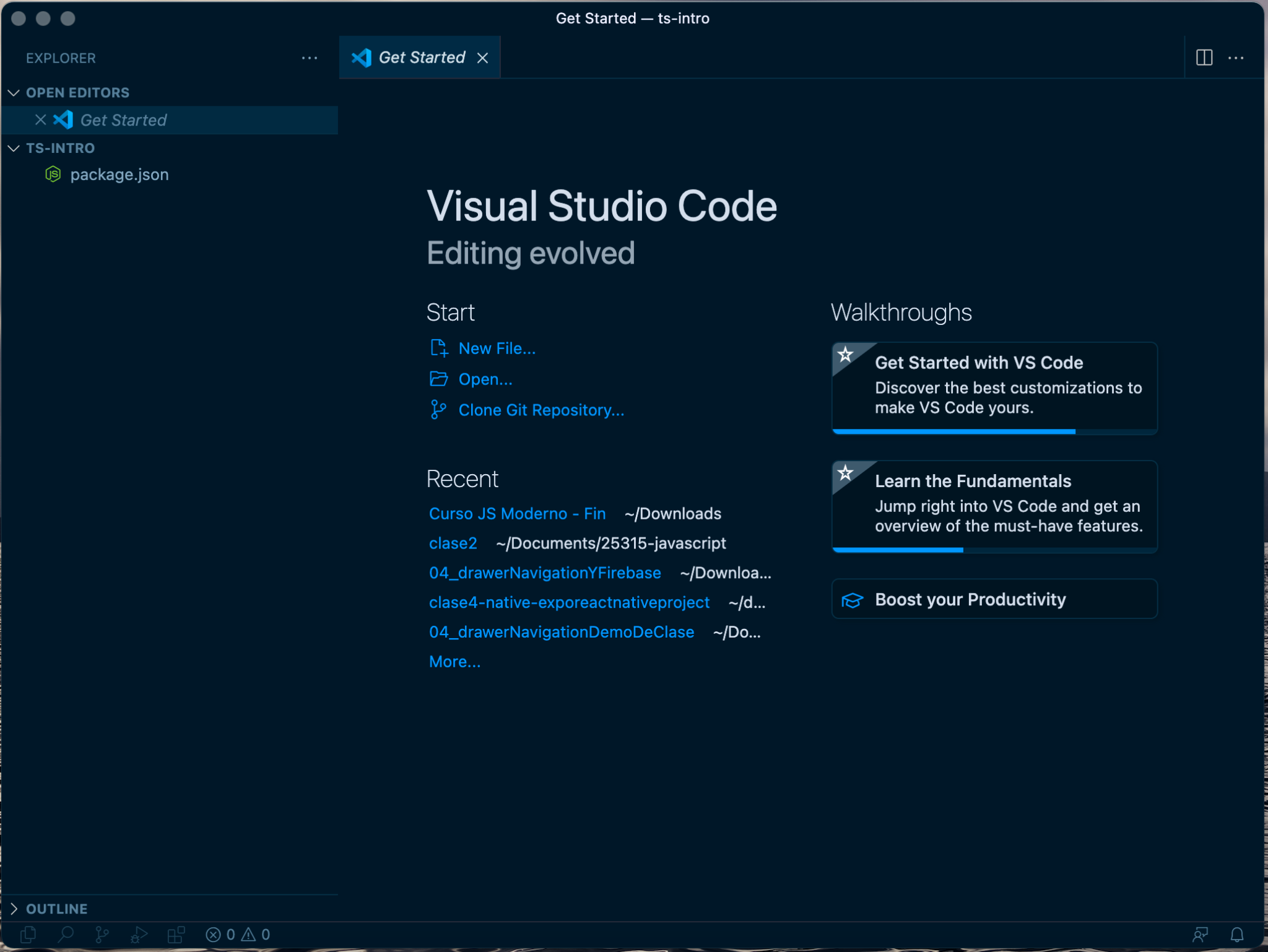


**ABRIMOS EL EDITOR**



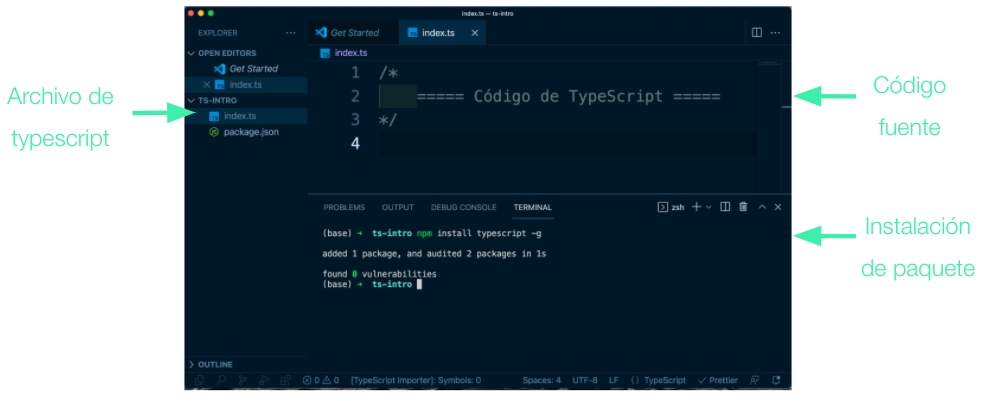
Iniciamos Visual Studio Code. Es decir, con el comando code . abrimos un editor en la carpeta donde estamos posicionados.

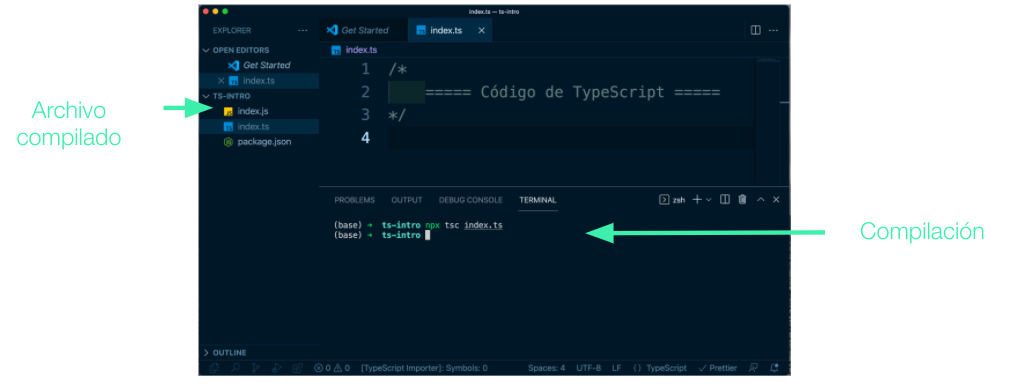
**VISUAL STUDIO CODE**

Archivo package.json   
generado con 

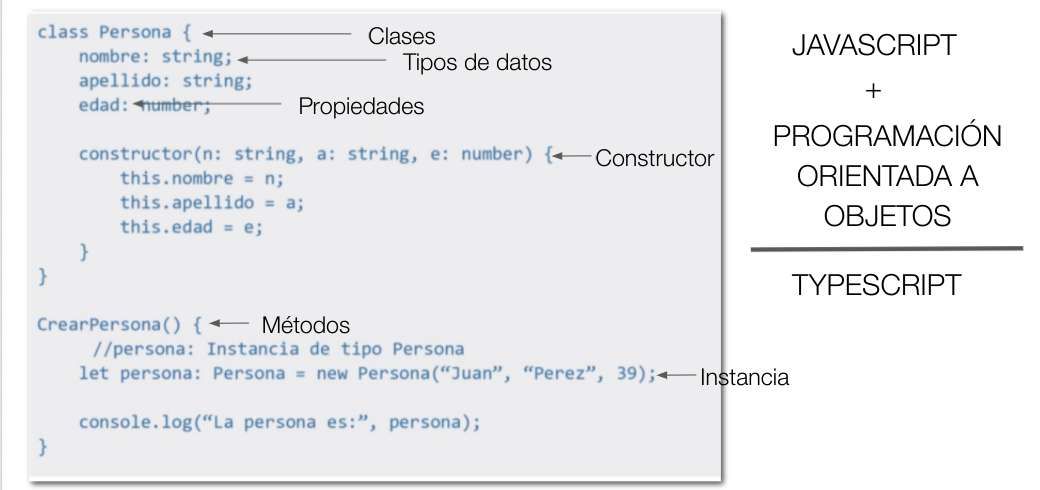
npm init→

**ESCRITURA DE CÓDIGO**

****



CARACTERÍSTICAS DEL LENGUAJE



**IMPORTANTE EN TYPESCRIPT**

* Tipos básicos
* Objetos, Arrays, Interfaces
* Clases
* Tipos genéricos
* Decoradores

**TYPADO**

Typescript tiene la capacidad de trabajar con tipado de datos de dos formas

1. Explícito: define una sintaxis para la creación de variables con tipo de dato

**nombreVariable = Tipo De Dato**

1. Inferido: puede deducir en función del valor

**nombreVariable = Valor**

**TIPOS PRIMITIVOS**



Referencia: [TypeScript](https://www.typescriptlang.org/)

**TIPOS BÁSICOS**

let nombre: string;

let edad: number | string;

let vive: boolean;

hp = 'FULL';

console.log(nombre, hp);

**ARRAYS**

Al igual que en Javascript, TypeScript permite definir Arrays para un conjunto de valores y se utilizan las notaciones de:

**[ ]**

let habilidades: string[] = ['Bash','Counter', 'Healing'];  
**Array <tipo>**

let pictures: Array<string>;

Pictures = ['Sunset','Vacation', 'Landscape']   
**INTERFACES**

Las interfaces en TypeScript constituyen una forma poderosa de definir “contratos” para el código que vaya a implementarlas.



interface Personaje {

nombre: string;

hp: number;

habilidades: string[];

puebloNatal?: string;

}

**OBJETOS**

const personaje: Personaje = {

nombre: 'Strider',

hp: 100,

habilidades: []

}

**object vs Object**

Object es un tipo de dato no primitivo.

👀 Cuidado al declarar a una variable con el tipo object porque no es lo mismo que crear un Object nativo de JS

¡Recuerda! 💡

Si tenemos un objeto declarado con el object de TS no podremos acceder a sus atributos mientras que si lo hacemos regularmente como en vanilla JS sí podremos.

os.

**CLASE**

class Heroe extends PersonaNormal {

constructor(

public alterEgo: string,

public edad: number,

public nombreReal: string

) {

super( nombreReal, 'New York, USA' );

}

}

Las clases y la POO, se pueden conectar las diferentes entidades y relacionarlas. Es decir, una clase es la abstracción de un conjunto de objetos.

**TIPOS GENÉRICOS**

Es una utilidad de TypeScript muy relacionada con su sistema de tipado y no existe en Javascript.

Los genéricos son plantillas de código que pueden definir y reutilizar en todo el código base. Proporcionan una manera de indicar a las funciones, clases o interfaces qué tipo quiere usar al llamarlas.

Referencia: Definición de genéricos en TypeScript | [docs.microsoft.com](https://docs.microsoft.com/es-es/learn/modules/typescript-generics/)

**DECORADORES**  
Son una característica exclusiva de typescript, por lo que cuando son compilados a es5, se crean funciones que expanden las clases de forma diferente.

El objetivo de los decoradores es cambiar las clases cuando son definidas.

function queTipoSoy<T>(argumento: T) {

return argumento;

}

let soyString = queTipoSoy('Hola Mundo');

let soyNumbero = queTipoSoy( 100 );

let soyArreglo = queTipoSoy( [1,2,3,4,5,6,7,8,9,10] );

let soyExplicito = queTipoSoy<number>( 100 );

DECORADORES

Son una característica exclusiva de typescript, por lo que cuando son compilados a es5, se crean funciones que expanden las clases de forma diferente.

El objetivo de los decoradores es cambiar las clases cuando son definidas.

Para conocer más ejemplos de decoradores, ingresa [aquí](https://www.typescriptlang.org/docs/handbook/decorators.html#class-decorators).

function classDecorator<T extends { new (...args: any[]): {} }>(constructor: T) {

return class extends constructor {

newProperty = "new property";

hello = "override";

};

}

@classDecorator

class MiSuperClase {

public miPropiedad: string = 'ABC123';

imprimir() { console.log('Hola Mundo') }

}

const miClase = new MiSuperClase();

VARIABLES

Una variable es un espacio reservado de memoria que utilizaremos para trabajar con datos en un programa, ya sea para guardar datos como para leer esos datos guardados 📂.

🚀 IMPORTANTE: todas las variables en typescript deben tener un tipo de dato, ya sea primitivo (number, string...) o un dato abstracto (Object, interface creada…)  
let *variable* = *nombre\_variable:tipo* = 'valor';

Las variables deben definirse con los siguientes límites:

* No pueden tener espacios.
* No empezar con un número.
* No puede ser una palabra reservada (if, for, while…).

let *variable* = *nombre variable:tipo* = 'valor';

let *variable* = *1variable:tipo* = 'valor';

let *variable* = *if:tipo* = 'valor';

Typescript define unos tipos de datos primitivos para la declaración de las variables:



let *booleano* = *variable:boolean* = true | false;

let *numero* = *variable:number* = 1;

let *string* = *variable:string* = 'texto';

let *variable* = *variable:any* = 'todos los tipos';

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

let *varios* = *variable:number|string* = 1 o 'texto';  
  
**CLASES E INTERFACES**

**Clases**  
Podemos utilizar plantillas para la creación de objetos de datos según un modelo predefinido. Se utilizan para representar entidades o conceptos. Define un conjunto de variables y métodos para operar con dichos datos 🙌.

Es conveniente declarar un constructor para realizar una correcta inicialización de las variables.

class Persona {

nombre: string;

edad: number;

constructor(nombre: string, edad: number) {

this.nombre = nombre;

this.edad = edad;

}

imprimir() { console.log(`Nombre:${this.nombre}`);

let jose: Persona = new Persona(‘Jose’, 30);

jose.imprimir();

}

**Interfaces**

Declara una serie de propiedades que deben ser implementados por una o más clases. Normalmente se utilizan para definir “tipos” que no tenemos disponibles por defecto con los primitivos de angular 😉.

¿Cómo se declaran? Las interfaces en TypeScript se declaran de manera bastante similar a la de las clases, indicando la lista de propiedades y métodos que contendrán 🎯 Solo hay un detalle fundamental: las propiedades no pueden tener valores y los métodos no pueden tener código para su implementación.  
  
**Interfaces como tipo**

//interfaz como tipo

interface persona {

nombre: string;

edad: number;

nacimiento: Date;

aficiones: string[]; //array de aficiones

casado?: boolean; //? Indica que no es obligatorio

}

let jose: persona; //variable con el nuevo tipo

**Interfaces como implementación**interface sumergibleInterface {

tiempoMaxBajoElAgua: number;

profundidadMaxima: number;

repelerAgua(): void;

}

class relojSumergible implements sumergibleInterface {

tiempoMaxBajoElAgua = 1;

profundidadMaxima = 10;

repelerAgua() { console.log('El agua me resbala');}  
  
💡 Declaramos la clase relojSumergible que implementa la interfaz sumergibleInterface, lo que nos permite acceder a sus propiedades y métodos.  
  
SISTEMA DE MÓDULOS: IMPORT Y EXPORT  
  
**Módulos**  
A partir de ECMAScript 2015, JavaScript tiene un concepto de módulos. TypeScript lo comparte y se puede definir de la siguiente manera:

Un módulo puede contener clases, variables o funciones. Sólo son visibles en el ámbito del módulo a menos que usemos los siguientes términos:

IMPORT. Permite al módulo utilizar los componentes del módulo importado.   
EXPORT. Permite que el contenido del módulo sea visible fuera del ámbito del mismo.  
  
**EXPORTACIÓN E IMPORTACIÓN**  
  
Las declaraciones (variable, const, función, clase, etc.) se pueden exportar para ser importadas en otro módulo.

Tenemos dos tipos de exportación: nombrada y predeterminada.

Las importaciones nombradas permiten especificar el elemento a importar.

// persona.ts

export function saludo(name: string){

console.log(`Buenos dias ${name}!`);

}

export const nombreConstante: string = 'Jorge';

export const numerico: number = 0;  
  


import {saludo, nombreConstante} from "./persona.ts";

saludo(nombreConstante); // Hola Jorge!

La exportación predeterminada indica qué partes se importan por defecto.

// persona.ts

const edadDefault = 35;

export default edadDefault;



import edadDefault from "./persona";

console.log(edadDefault);

Otra funcionalidad en Typescript es reexportar declaraciones:  
//operador.ts

interface Operator {

eval(x: number, y: number): number;

}

export default Operator;



//suma.ts

import Operator from "./Operator";

export class Add implements Operator {

eval(x: number, y: number): number {

return x + y;

}

}

//persona.ts

export interface persona {

nombre: string;

edad: number;

}



//tratamientoPersona.ts

import { persona } from ‘./persona.ts’;

let alba: persona = {nombre: ‘Alba’, edad: 35};

¿QUIERES SABER MÁS? TE DEJAMOS MATERIAL AMPLIADO DE LA CLASE

[TypeScript in Visual Studio Code](https://code.visualstudio.com/docs/languages/typescript) | CodeLens

[What is TypeScript?](https://www.typescriptlang.org/) | TypeScript

Resumen de lo visto en clase hoy:

* Configuración de un proyecto
* Tipado, clases, interfaces, decoradores.
* Variables
* Sistema de módulos.