	<b>UNIVERSIDAD DON BOSCO</b> <b>FACULTAD DE INGENIERIA, ESCUELA DE COMPUTACION</b>
Ciclo II	<b>DISEÑO Y PROGRAMACION DE SOFTWARE MULTIPLATAFORMA</b> <b>Guía de Laboratorio N# 02</b> <b>“Introducción a Typescript”</b>

## I. RESULTADOS DE APRENDIZAJE

Que el estudiante:

- Configure el entorno de trabajo para poder iniciar a trabajar con Typescript
- Poner en práctica Typescript, para trabajar con Angular.

## II. INTRODUCCIÓN

JavaScript es uno de los lenguajes más populares, en parte porque ha evolucionado y mejorado a pasos agigantados en los últimos años.

Sin embargo, Javascript en algún punto fue un lenguaje que presentaba muchos problemas para bases de código grandes, aplicaciones de gran escala y proyectos con muchos años de desarrollo.

En 2012 en Javascript no había clases, ni módulos, el ecosistema carecía de herramientas que optimizaran el flujo de desarrollo, derivado precisamente por las carencias del lenguaje mismo.

2012 fue el año en que Typescript apareció (luego de 2 años de desarrollo), una solución de Microsoft para el desarrollo de aplicaciones con Javascript a gran escala, para ellos y para sus clientes. Steve Lucco y un equipo de más de 50 personas que incluía a Anders Hejlsberg, Lead Architect de C# y creador de Delphi y Turbo Pascal desarrollaron Typescript en Microsoft, un proyecto que originalmente se conoció como Strada.

Originalmente, productos como Bing y Office 365 despertaron en Microsoft la necesidad de una mejora a JavaScript que permitiera construir productos escalables.

**Typescript** es la solución a muchos de los problemas de JavaScript, está pensado para el desarrollo de aplicaciones robustas, implementando características en el lenguaje que nos permitan desarrollar herramientas más avanzadas para el desarrollo de aplicaciones.

## SUPERSET DE JAVASCRIPT

Typescript es un superset de JavaScript. Decimos que una tecnología es un superset de un lenguaje de programación, cuando puede ejecutar programas de la tecnología, Typescript en este caso, y del lenguaje del que es el superset, JavaScript en este mismo ejemplo. En resumen, esto significa que los programas de JavaScript son programas válidos de TypeScript, a pesar de que TypeScript sea otro lenguaje de programación.

Esta decisión fue tomada en Microsoft bajo la promesa de que las futuras versiones de EcmaScript traerían adiciones y mejoras interesantes a Javascript, esto significa que Typescript se mantiene a la vanguardia con las mejoras de JavaScript.

Además, esto permite que uno pueda integrar Typescript en proyectos existentes de JavaScript sin tener que reimplementar todo el código del proyecto en Typescript, de hecho, es común que existan proyectos que introduzcan tanto Typescript como JavaScript.

Por si fuera poco, uno de los beneficios adicionales de esta característica del lenguaje, es que pone a disposición el enorme ecosistema de librerías y frameworks que existen para JavaScript. Con Typescript podemos desarrollar aplicaciones con React, Vue, Angular, etc.

## TIPADO ESTÁTICO

La principal característica de Typescript es el tipado estático. Decimos que un lenguaje es de tipado estático cuando cumple con estas características principales:

- Las variables tienen un tipo de dato.
- Los valores sólo se pueden asignar a variables del tipo correspondiente.

```
let edad : number; //Asignamos el tipo number para la variable edad
edad = 20; // La variable ahora sólo puede asignar valores del tipo number
```

A partir de estas dos características principales, se derivan algunas otras, como, por ejemplo:

- Interfaces
- Genéricos
- Casting de datos (conversión de tipos)
- Argumentos con tipo
- Tipo de retorno para las funciones
- Mucho más

El contraste de estos lenguajes son los de tipado dinámico, como JavaScript, estos lenguajes suelen ser mucho más flexibles, lo que nos permite escribir código menos verboso.

Por otro lado, los lenguajes de tipado estático se prestan a la implementación de herramientas de desarrollo más avanzadas, como:

- Autocompletado de código
- Recomendación de qué argumentos recibe una función
- Recomendación de qué tipo retorna una función
- Auto documentación del código
- Mejor análisis para detectar errores

```
class Speaker{
  @Auditable
  n() { console.log(20); }
}
let speaker : Speaker = new Speaker();
speaker.n().
  Auditable(clsPro: any, methodName: string, descriptor?: any
): void
Auditable()
```

## POR QUÉ TYPESCRIPT

La industria de la programación es increíblemente diversa, incluso si te especializas en algún área de la programación, todos los días hay algo nuevo que aprender, una nueva tecnología, un nuevo enfoque para solucionar los problemas, etc.

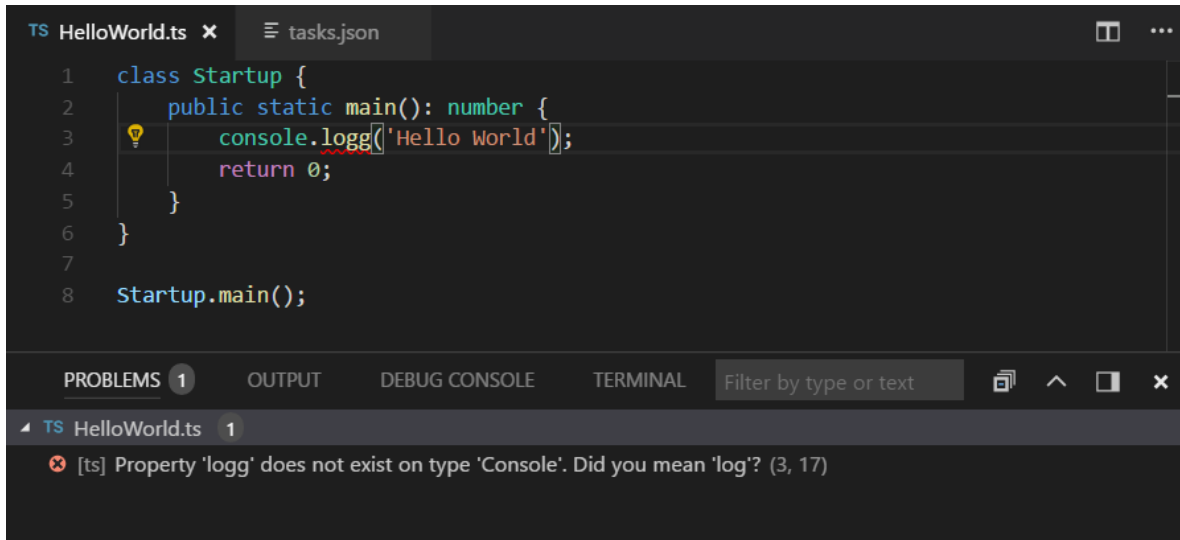
De vez en cuando llega una tecnología que deslumbra tus ojos, de la que inmediatamente te enamoras e incluso te preguntas, ¿cómo es que podías trabajar sin esta tecnología? Typescript es esa clase de tecnología.

Cuando escribes un lenguaje con la intención de que éste se preste para el desarrollo de herramientas para los desarrolladores ¿qué obtienes? Un lenguaje de programación con una experiencia de desarrollo superior a otros.

**Desarrollar en Typescript** es increíble, prueba de ello es la cantidad de equipos de desarrollo que han integrado la herramienta, como sustituto del uso de JavaScript, entre ellos tenemos:

- Google
- Microsoft
- Basecamp
- Lyft
- Miles de empresas más

## VISUAL STUDIO CODE



Visual Studio Code, un editor de texto de Microsoft, integra decenas de herramientas muy interesantes para TypeScript.

## FRAMEWORKS JAVASCRIPT

El equipo que desarrolla alguno de los open sources más populares, usan TypeScript para el desarrollo de dichas librerías. Un par de ejemplos muy populares son Angular y Stimulus.

Esto se traduce en que la mayor parte de la documentación y ejemplos de estos frameworks, principalmente Angular, se escribe con TypeScript. Eventualmente esto significa que las aplicaciones que se desarrollan usando estas librerías se hacen con TypeScript.

Angular es quizás el ejemplo más destacado, prácticamente todas las aplicaciones en Angular se escriben usando TypeScript, si quieres entender el framework a fondo, necesitas saber TypeScript.

Estas decisiones no se toman a la ligera, los equipos que desarrollan estos frameworks aprovechan las características de TypeScript para la implementación de estos frameworks y el desarrollo de aplicaciones web.

## TYPESCRIPT ES UN SUPERSET DE JAVASCRIPT

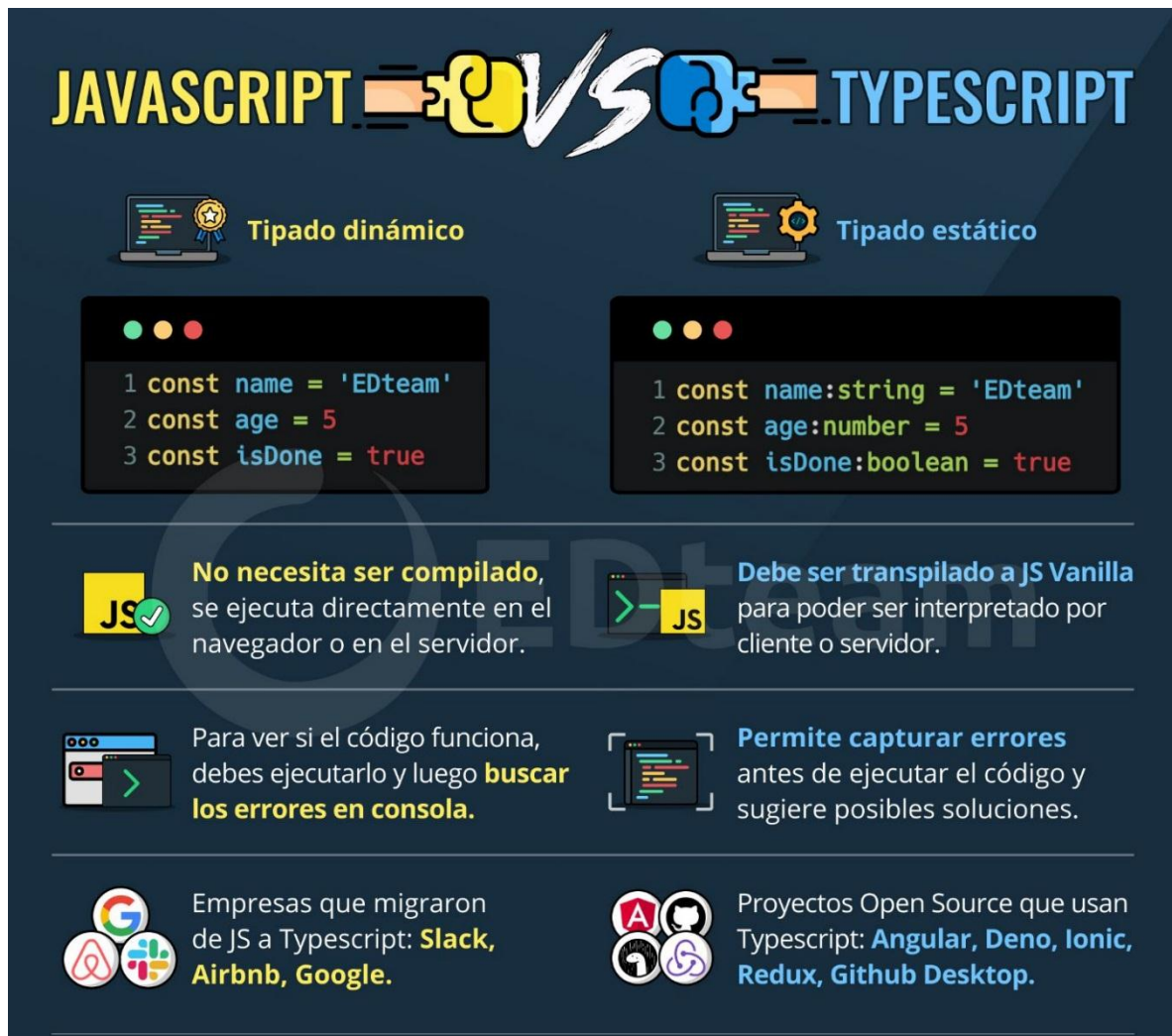
Recuerda que TypeScript es un superset de JavaScript, eso significa que en todos los casos en los que puedes utilizar JavaScript, también puedes usar TypeScript.

## CONCLUSIÓN

Aprender Typescript te introduce a un lenguaje de programación increíble, con herramientas para desarrollo aún mejores. Particularmente **Visual Studio Code** ofrecerá un número interesante de mejoras a tu flujo de desarrollo.

Además, algo de lo increíble de usar Typescript es que puedes usarlo en diferentes entornos y para el desarrollo de Aplicaciones nativas, híbridas, web, de escritorio e incluso servidores web.

## JavaScript vs TypeScript



The infographic is titled "JAVASCRIPT VS TYPESCRIPT" in large, stylized letters. It compares the two languages across several categories:

- Tipado dinámico (Dynamic Typing):** Represented by a laptop icon with a star. It shows a code snippet: 

```
1 const name = 'EDteam'
2 const age = 5
3 const isDone = true
```
- Tipado estático (Static Typing):** Represented by a laptop icon with a gear. It shows a code snippet: 

```
1 const name:string = 'EDteam'
2 const age:number = 5
3 const isDone:boolean = true
```
- Execution:** JavaScript is described as "No necesita ser compilado, se ejecuta directamente en el navegador o en el servidor." (No need to be compiled, it runs directly in the browser or on the server). TypeScript is described as "Debe ser transpilado a JS Vanilla para poder ser interpretado por cliente o servidor." (Must be transpiled to Vanilla JS to be interpreted by client or server).
- Error Handling:** JavaScript requires running the code and then "buscar los errores en consola." (search for errors in the console). TypeScript "Permite capturar errores antes de ejecutar el código y sugiere posibles soluciones." (Allows capturing errors before running the code and suggests possible solutions).
- Adoption:** JavaScript is used by companies that migrated from JS to TypeScript: Slack, Airbnb, Google. TypeScript is used by Open Source projects: Angular, Deno, Ionic, Redux, Github Desktop.

### III. MATERIALES Y EQUIPO

Para la realización de la guía de práctica se requerirá lo siguiente:

#### Hacer la Instalación de:

<https://code.visualstudio.com/> (Versión 1.47.3)

<https://nodejs.org/es/> (Versión node-v12.18.3)

Tutorial de TypeScript en Visual Studio Code

<https://code.visualstudio.com/docs/typescript/typescript-tutorial>

#### Instalando TypeScript

<https://www.npmjs.com/>

Para poder usar TypeScript necesitas tener instalado previamente Node.js, y después en nuestra terminal ejecutar el comando:

```
npm install -g typescript
```

Usando tu editor de texto Visual Studio Code para ir creando los archivos de TypeScript recordando que la extensión es .ts.

Datos básicos en TypeScript

#### Imprimir en consola

Para imprimir datos en consola no es diferente a JavaScript:

```
console.log("Mi mensaje");
```

#### Variables y valores primitivos

Las variables tienen la palabra reservada var, pero el tipo de dato se especifica usando :tipo. Tenemos 3 tipos de datos primitivos: string, number y boolean. Ejemplo:

```
var full_nombre:string = "jorge cano";  
var age:number = 27;  
var developer:boolean = true;
```

## Arreglos

Los arreglos se declaran con la palabra `Array<Tipo>`. Donde el tipo son los valores primitivos que vimos anteriormente. Recuerda que solo pueden existir en el Array valores del tipo antes especificado. *Ejemplo:*

```
var skills:Array<String> = ['JavaScript','TypeScript','Angular'];
```

También podemos declararlos de la forma:

```
var numberArray:number[] = [123,123,1213,1231];
```

## Enumerables

Los Enumerables son objetos que enumeran los elementos dentro de él. Se usa principalmente para definir roles o configuraciones y evitar el uso de constantes. Tienen la palabra reservada `enum` y para el nombre es recomendable que se encuentre todo en mayúscula. *Ejemplo:*

```
enum ROLE {Employee, Manager, Admin, Developer }
```

*Entonces podemos asignarle un valor de Role a cualquier variable:*

```
var role:Role = Role.Employee
```

## Funciones

Las funciones son bloques de código que esperamos ejecutar de manera recurrente. Tienen la palabra reservada `function` y tenemos múltiples formas de declararlas según las necesidades, por ejemplo: a) Función que no retorne nada y que no reciba parámetros:

```
functionhello():void {  
}
```

b) Función que no retorne nada y que reciba un parámetro del tipo string:

```
functionsetName(name:string):void{  
}
```

c) Función que retorne un dato tipo string y que reciba 2 parámetros del tipo string:

```
functionsetName(name:string, surName:string ):string {
```

```
return"string";  
}
```

Es importante de observar que seguimos necesitando indicar que tipo de dato es que el vamos a recibir y/o regresar. **Scope de una variable y la palabra “this”** Las variables que se encuentran dentro de una función no pueden ser accedidas desde otras partes fuera de ella. Por ejemplo:

```
// Teniendo la función  
functionsetName(name:string):string{  
  var variableInterna:string = "Uriel";  
  return"Hola" + name;  
}  
// No se puede acceder a la variable Interna console.log("Hola" + variableInterna );  
Sin embargo, podemos acceder a las variables globales que se encuentren fuera de una  
función desde cualquier parte, inclusive desde en interior de una función.
```

```
// Teniendo la misma función pero con la variable fuera de ella  
var variableExterna:string = "Uriel";  
  
functionsetName(name:string):string{  
  // Accedemos a la variable externa usando "this" this.variableExterna = name;  
  return"Hola" + name;  
}  
  
// Desde aquí podemos igualmente acceder a la variable  
console.log("Hola" + variableExterna );
```

**Pasando de .ts a .js** Al instalar TypeScript, se agregan una serie de herramientas que nos ayudan a ‘transpilar’ nuestro código. Solo necesitas ejecutar el comando

```
tsc "NombreDelArchivo".ts
```

Y al terminar, en el mismo directorio, encontrarás el .js transformado con el mismo nombre de tu archivo de TypeScript, y ya listo para usar para cualquier navegador.

## TypeScript CLI

**TypeScript CLI** es la herramienta que nos va a ayudar a ‘transpilar’ nuestros archivos de TypeScript a JavaScript. Podemos hacer entre otras cosas ‘transpilaciones’ globales, de un solo archivo, elegir la versión de ECMAScript que queremos, etc. Recordando la instalación:

```
npm install -g typescript
```



Para pasar de .ts a .js sólo necesitas ejecutar el comando

```
tsc "NombreDelArchivo".ts
```

Y al terminar, en el mismo directorio, encontrarás el .js transformado con el mismo nombre de tu archivo de TypeScript, y ya listo para usar para cualquier navegador.

### ¿Cuáles son las diferencias entre compilar y transpilar?

**Compilar** es generar código ejecutable por una máquina, que puede ser física o abstracta como la máquina virtual de Java.

**Transpilar** es generar a partir de código en un lenguaje código en otro lenguaje. Es decir, un programa produce otro programa en otro lenguaje cuyo comportamiento es el mismo que el original.

Bajo esta definición se puede argumentar que todos los compiladores en realidad transpilan a lenguaje máquina, pero en la práctica no se usa esta extensión de la definición, ya que los “transpilars” existentes generan código de muy alto nivel en comparación con el lenguaje máquina.

### Objetos en TypeScript

Un objeto es una abstracción de algo que existe en el mundo “real”, el cual llevamos todas sus características a código. Estos objetos se crean por algo llamado “clases”. En TypeScript tenemos la palabra reservada class. Ejemplo:

```
class Persona {  
  first_name:string;  
  last_name:string;  
  constructor(_first_name?:string, _last_name?:string) {  
    this.first_name = "_first_name";  
    this.last = "_last_name";  
  }  
}
```

Es importante decir que una clase siempre tiene un constructor y ese constructor es lo primero que se va a llamar por defecto cuando hagas una instancia de esa clase. Si tú no lo implementas, por defecto es el constructor más sencillo:

```
constructor(){  
}
```

Sin embargo podemos tener parámetros obligatorios u opcionales, los opcionales que se distinguen por tener un “?”.

```
constructor(_first_name?:string, _last_name?:string)
```

```
this.first_name = "_first_name";
this.last = "_last_name";
}
```

Para hacer una instancia de la clase, podemos crear una nueva variable o constante e indicar la clase mediante la palabra *new*:

```
let personaUno = new Persona();
let personaDos = new Persona("Jorge");
let personaTres = new Persona("Jorge", "Cano");
```

¿Qué pasa si los parámetros de mi constructor no son opcionales? **R.** En el caso de que tengamos parámetros obligatorios, no se podrá crear una instancia de la clase sin mandar esos datos en el constructor:

```
//Teniendo en cuenta:
constructor(_first_name:string, _last_name:string)
this.first_name = "_first_name";
this.last = "_last_name";
}
// Esto ya no se podrá ser una instancia de esa claselet personaDos = new Persona("Jorge");
//Esta será una instancia de esa claselet personaTres = new Persona("Jorge", "Cano");
```

## Interpolation

Interpolation es la forma en que mezclamos variables justo con los strings. Acompañado de comillas francesas, podemos acceder al valor de una variable de forma directa desde ese string. Ejemplo:

```
var a:string = "Uriel";

var b = `Saludos a ti ${this.a},`;

console.log(b);`
```

Otra de las ventajas de usar las comillas francesas es que respeta cada espacio y cada salto de línea que insertamos, tal como un template:

```
getSaludo():string{
    let emojis = '(-■_■)';
    return`Saludos
    ${this.last_name}, ${this.first_name}
    Le enviamos un saludo desde la consola!
    ${emojis}
    `;
}
```

```
}
```

## Interfaces

A diferencia de las Clases, las interfaces no se instancian mediante constructores, si no mediante un objeto JSON. Tienen la palabra reservada `interface`. Ejemplo:

```
interface MyPersona {  
  first_name: string;  
  last_name: string;  
  twitter_account?: string;  
}
```

De igual manera podemos tener campos obligatorios y opcionales. Para hacer uso de una interfaz, incluimos en el tipo de dato el nombre de la interfaz y le asignamos un objeto:

```
let personaUno: MyPersona = {  
  first_name: 'Jorge',  
  last_name: 'Cano',  
  twitter_account: '@jorgeucano'  
}
```

## Shapes

Los Shapes son muy parecidos en implementación a las clases y siguen siendo objetos. La única diferencia es que no dependemos de los constructores para asignar valores a las variables internas de las clases, si no que asignamos directamente el valor después de la creación de la instancia:

```
class Person {  
  first_name: string;  
  last_name: string;  
  twitter_user: string;  
  
  constructor() {  
    this.first_name = "Jor";  
    this.last_name = "Ca";  
    this.twitter_user = "@jorgeucano";  
  }  
  
  setLastName(last_name: string) {  
    this.last_name = last_name;  
  }  
}
```

```
var myPerson = new Person();
myPerson.first_name = "Jorge";
myPerson.setLastName("Cano");
console.log(myPerson);
```

Mientras las variables dentro de la clase sean públicas, podemos hacer uso de los `shaves`, de otro modo, se harían por `getters` y `setters`.

## Decorators

Los `decorators` son algo que se usan mucho en Angular pero que también tenemos en TypeScript. Los `decorators` nos permiten agregar una funcionalidad extra a algo, como por ejemplo a métodos o miembros de una clase. La estructura más simple de un `decorator` es:

```
Function color(value: string){ // Así definimos el decorator

fabric return function(target){

    // Este es el decorator, prácticamente regresamos una una función con la funcionalidad, se
    pide el objetivo

    // Aquí es donde modificamos el objetivo con el valor pedido desde el decorator

    }
}
```

Particularmente, los `decorators` de método necesitan de 3 parámetros en su definición:

- **Target:** es método el cuál vamos a aplicar el `decorator`
- **Key:** es nombre del método el cuál vamos a aplicar el `decorator`
- **Value:** es `Indefinido` en otros casos. Un ejemplo:

```
class Greeter {

    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }

    @enumerable(false)
    greet() {
        return "Hey, " + this.greeting;
        // return
    }
}
```

```

}

function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}

let gree = new Greeter("Soy el mensaje");

console.log(gree.greet());

```

Lo que va a hacer nuestros decorator `@enumerable(false)` es verificar si el tipo de dato no es enum. De serlo, no podrá implementar el método `greet()`. Por último, con TypeScript podemos tener un archivo de configuración para aprovechar todas las ventajas al momento de hacer la ‘transpilación’, entre muchas cosas tenemos:

```

{
  "compilerOptions": {
    "target": "ES5",
    // Definir en que versión de JS nos gustaría convertir el .ts"

    "experimentalDecorators": true,
    // Activar funciones experimentales, en este caso decorators

    "module": "system", // Usar el modulo system
    "noImplicitAny": true, // Habilitar el uso de implícitos Any
    "removeComments": true, // Remover todos los comentarios en nuestros archivos
    "preserveConstEnums": true, // Manejar los enumerables como constantes, esto es muy
recomendado
    "OutFile": "./tsc.js", // Donde vamos a tener el archivo resultante
    "sourceMap": true // Para ver un log de la transpilación
  },
  "include": [
    "**" // Ver que archivos vamos a compilar, usar * indica que serán todos
  ],
  "exclude": [
    "node_modules" // Agregar los archivos que no queremos incluir en la transpilación "
    "**/*.spec.ts"
  ]
}

```

Y ya guardado este archivo como **tsconfig.json** podemos ejecutar en la terminal el comando **tsc** para realizar los cambios. Durante la explicación, tuvimos diferentes archivos .ts, y notarás que, con la ayuda de la configuración, resultó un solo archivo .js que contiene todos los ejercicios ahora juntos.

## Ejercicios para realizar:

- 1- Crear la clase Rombo, la cual debe tener dos propiedades: DiagonalVertical y DiagonalHorizontal. añadiremos un constructor al que le pasaremos los valores anteriores cuando instanciamos el objeto. Y también debe de tener un método que calcule el area, que será la multiplicación de DiagonalVertical \* DiagonalHorizontal. Este método devolverá un número.
- 2- Crear una interface en TypeScript a partir de este código JavaScript:  

```
var spiderman = {  
  nombre: "Peter parket",  
  poderes: ["trepar", "fuerza", "agilidad", "telas de araña"]  
};
```
- 3- Crear una clase Empleado, con las propiedades nombre, salario, añadiremos un constructor al que le pasaremos los valores anteriores cuando instanciamos el objeto. Y también debe de tener un método que calcule deducciones salariales, Este método devolverá el salario después de los descuentos.
- 4- Crear una clase Calculadora, con las propiedades numero1, numero1, Le añadiremos un constructor al que le pasaremos los valores anteriores cuando instanciamos el objeto. Y también debe de tener un método operaciones básicas (+ , - , \* , /), Este método debe imprimir en pantalla todas las operaciones realizadas.