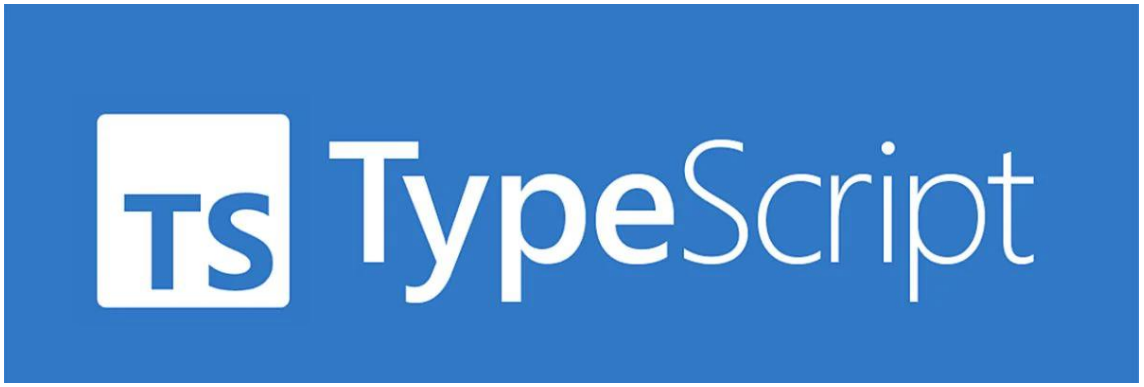
2023



Tema 4: TypeScript

TYPESCRIPT

STUDIUM

www.grupostudium.com
informacion@grupostudium.com
954 539 952

1 Introducción

TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipos estáticos y objetos basados en clases.

El código TypeScript es *transcompilado*, es decir se transforma en JavaScript y luego se puede integrar en una página HTML.

2 Integrar TypeScript en un proyecto web

2.1 npm

La forma más rápida para decirle a nuestro entorno de desarrollo que queremos utilizar TypeScript en nuestro proyecto, es usar el *administrador de paquetes* oficial de Node.js llamado npm.
npm permite la instalación de paquetes y dependencias confiando en repositorios oficiales.

Para ello primero debemos tener instalados Node.js y npm en nuestro ordenador

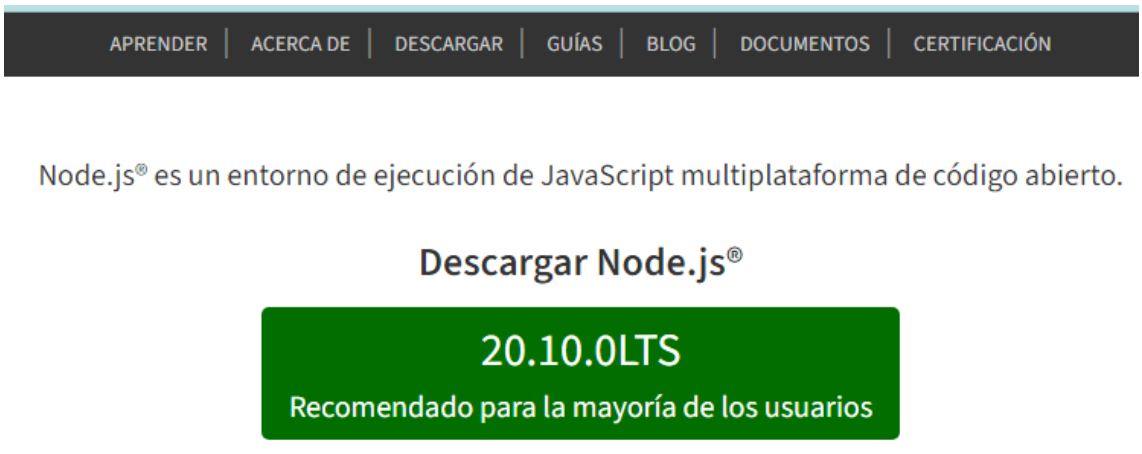
- Tener instalado Node.js: entorno de ejecución de JavaScript que se utiliza para instalar y ejecutar aplicaciones basadas en JavaScript en el servidor.
- Tener instalado npm: administrador de paquetes oficial de Node.js que se utiliza para instalar y administrar paquetes de Node.js.

Una vez que tengamos instalados Node.js y npm, podemos instalar TypeScript usando el comando de instalación de npm que veremos a continuación.

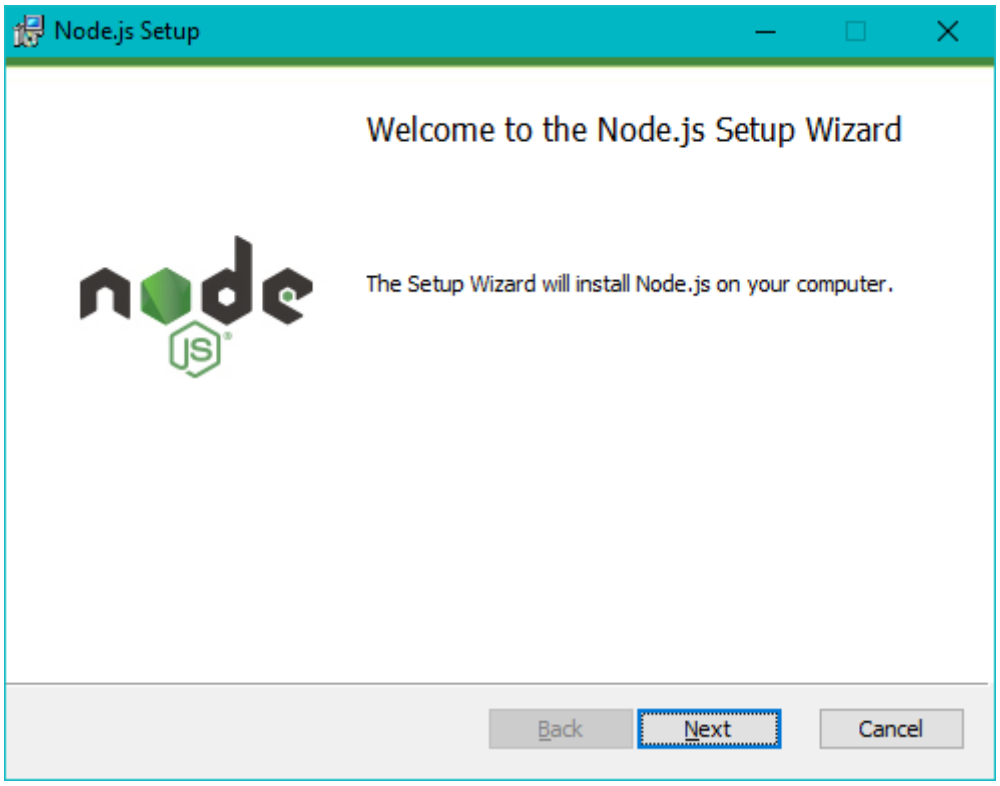
2.1.1 Instalar Node.js

TypeScript necesita Node.js para ejecutarse, por ello lo primero que hay que hacer es descargar e instalar Node.js desde su sitio web oficial:

https://nodejs.org/en



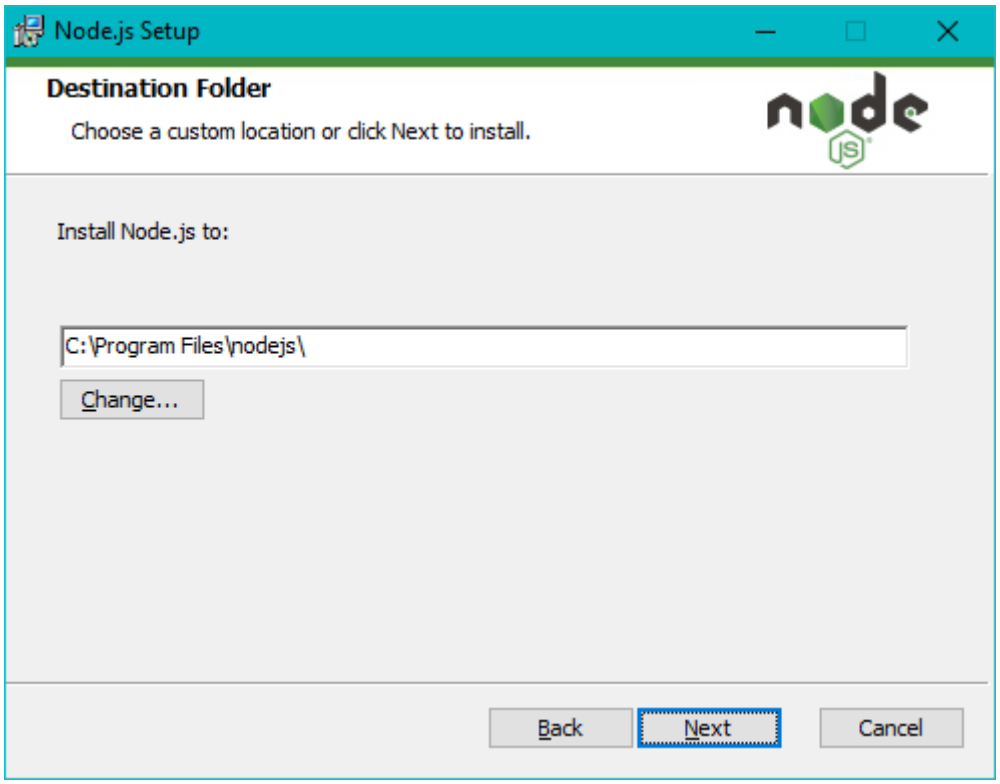
Se descarga el fichero node-v20.10.0-x64.msi que debemos instalar.



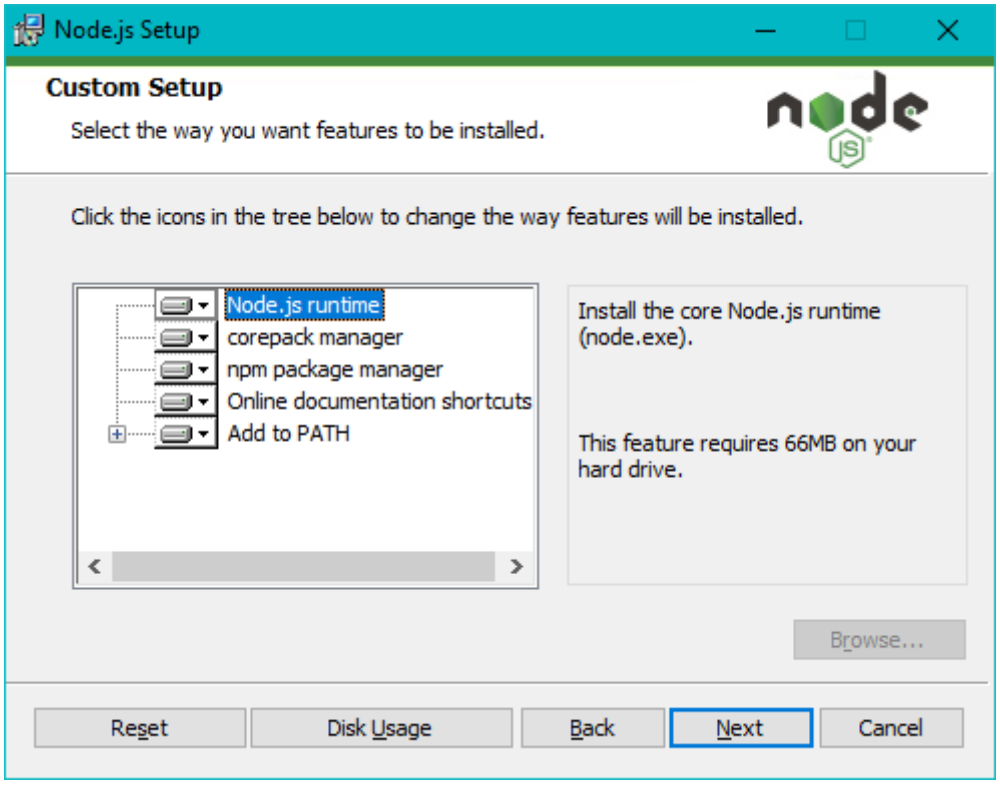
Next.

Aceptamos los términos de licencia y Next.

www.grupostudium.com
informacion@grupostudium.com
954 539 952

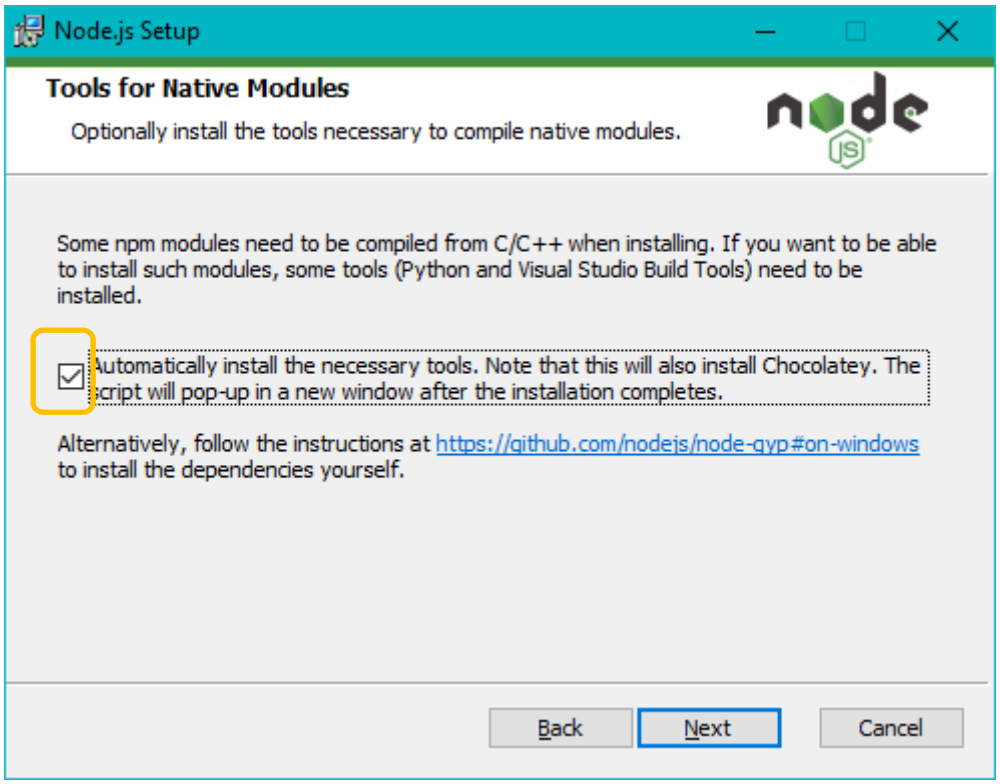


Next.

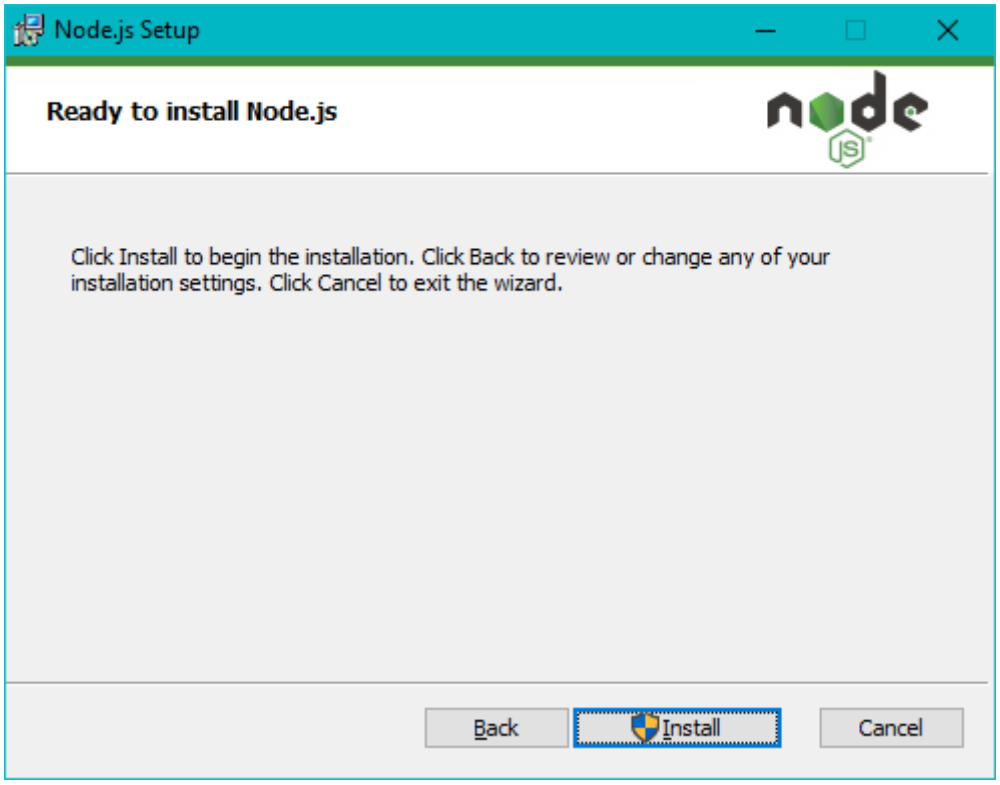


Next.

www.grupostudium.com
informacion@grupostudium.com
954 539 952

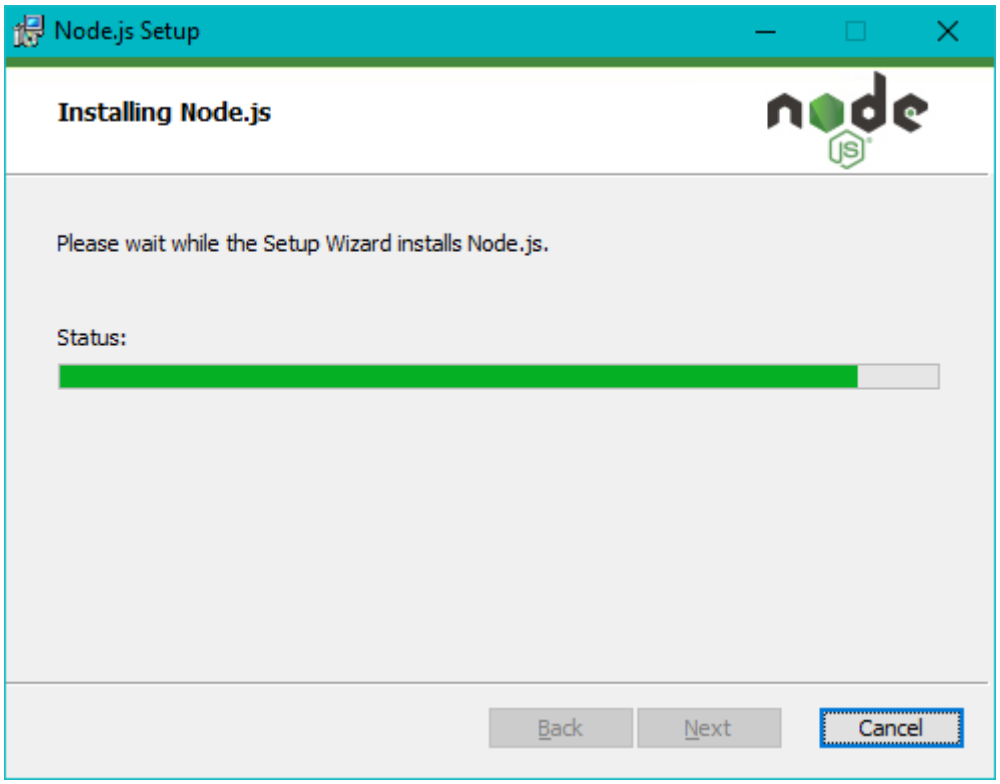


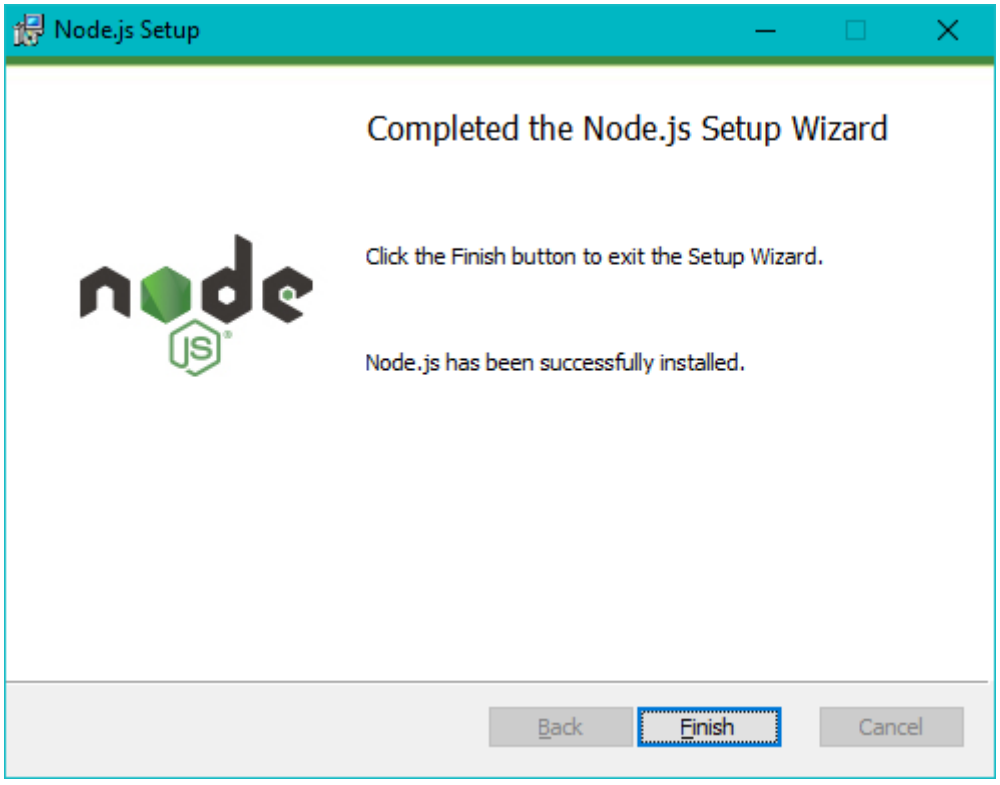
Next.



Install.

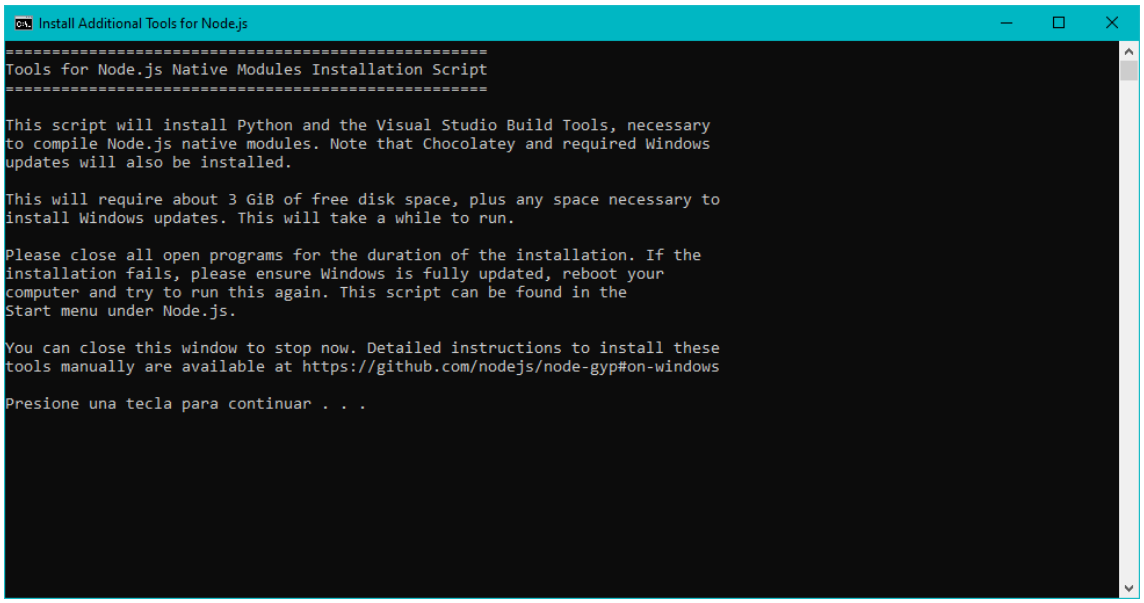
www.grupostudium.com
informacion@grupostudium.com
954 539 952



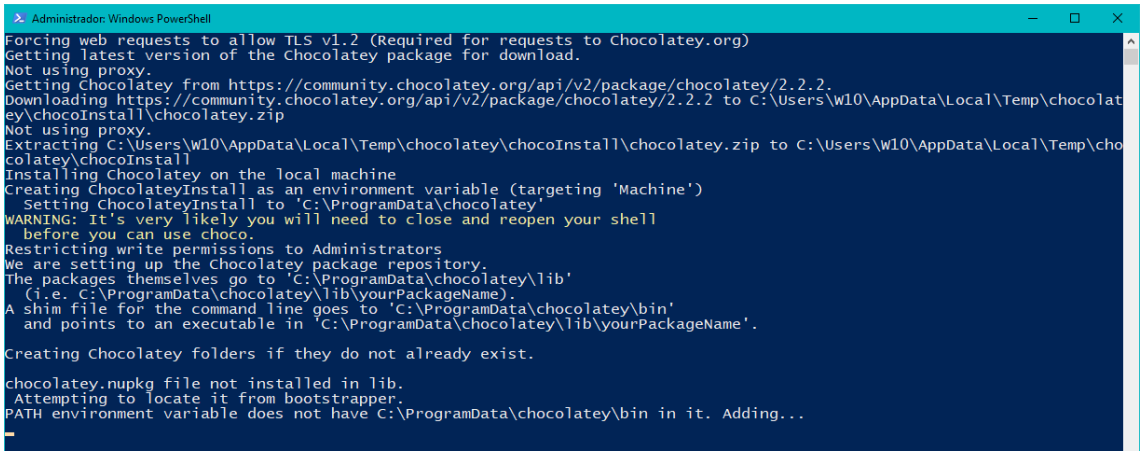


Finish.

www.grupostudium.com
informacion@grupostudium.com
954 539 952



Enter, Enter. Se abre una ventana de Windows PowerShell en la que continúa la instalación:



Una vez haya finalizado la instalación de Node.js verificamos su instalación abriendo una ventana de terminal y escribiendo el comando:

```
node -v
```

Si se muestra la versión de Node.js instalada, significa que la instalación se realizó correctamente..

2.1.2 Instalar TypeScript

Abrimos una ventana de terminal desde nuestro Visual Studio Code y ejecutamos el siguiente comando:

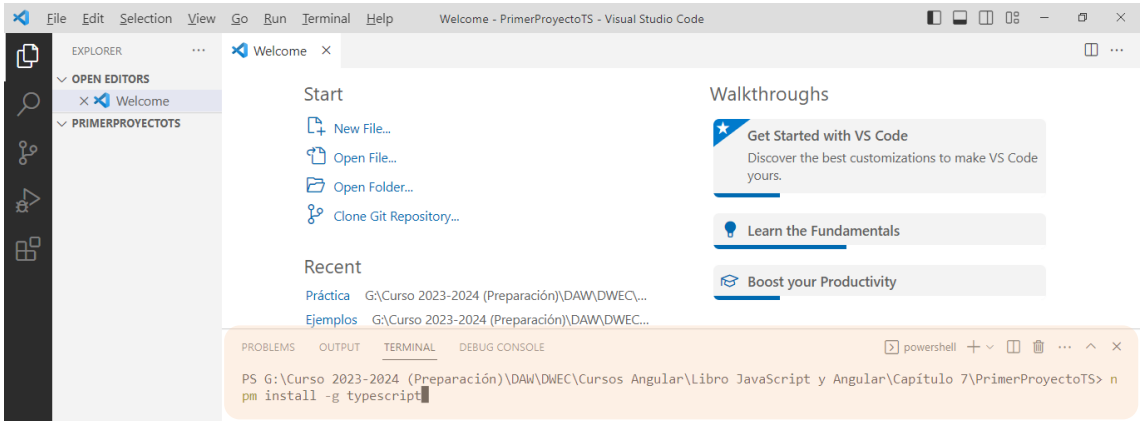
```
npm install -g typescript
```

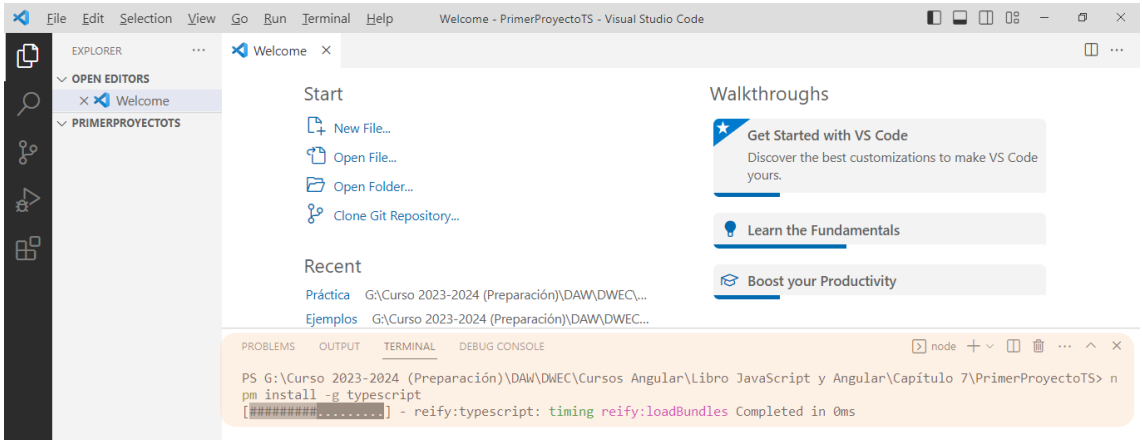
Este comando instala TypeScript de forma global en nuestro ordenador.

En este tema estudiaremos las funcionalidades útiles de TypeScript para desarrollar con el framework Angular.

Ejemplo: Creamos una carpeta con el nombre del proyecto PrimerProyectoTS, ejecutamos Visual Studio Code y abrimos esta carpeta desde el menú File – Open Folder. Abrimos un terminal desde el menú Terminal – New Terminal, observamos que se ha abierto en la carpeta PrimerProyectoTS y desde aquí, el raíz de nuestro proyecto, ejecutamos el comando de instalación de typescript para poder desarrollar nuestra aplicación usando TypeScript:

```
npm install -g typescript
```

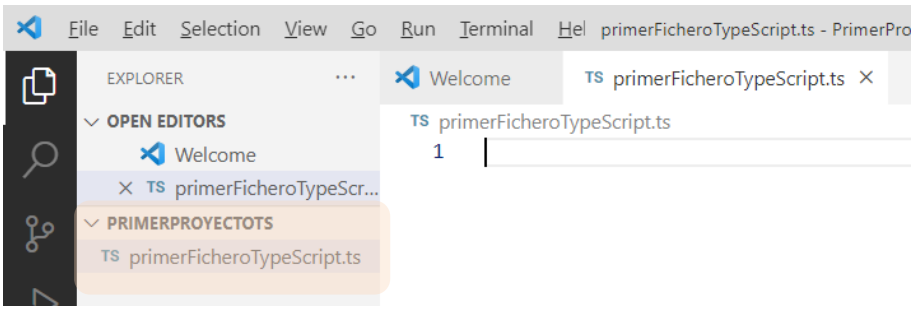




2.2 Creación de un archivo TypeScript

Visual Studio Code permite la creación de un archivo TypeScript de forma muy sencilla, ya que solamente tenemos que crear un nuevo archivo con la extensión **.ts**.
Ejemplo: Creamos el fichero primerFicheroTypeScript.ts en la carpeta de nuestro proyecto.

www.grupostudium.com
informacion@grupostudium.com
954 539 952



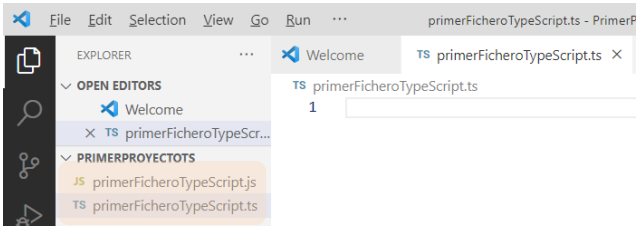
Para transcompilar este archivo e integrarlo en una página HTML, el comando que tenemos que ejecutar en una terminal de Windows es:

```
tsc primerFicheroTypeScript.ts
```

Al ejecutar este comando, se crea un fichero de extensión .js con el mismo nombre del fichero .ts que teníamos creado: primerFicheroTypeScript.js

Este fichero primerFicheroTypeScript.js se puede integrar fácilmente en una página web, a través de sus etiquetas <script></script>.

```
PS G:\Curso 2023-2024 (Preparación)\DAW\DWEC\Cursos Angular\Libro JavaScript y Angular\Capítulo 7\PrimerProyectoT
S> tsc primerFicheroTypeScript.ts
PS G:\Curso 2023-2024 (Preparación)\DAW\DWEC\Cursos Angular\Libro JavaScript y Angular\Capítulo 7\PrimerProyectoT
S>
```



Un archivo TypeScript transcompilado se puede lanzar directamente usando el siguiente comando node en un terminal:

```
node primerFicheroTypeScript.js
```

3 Tipado

3.1 JavaScript y los errores de tipado

JavaScript no está muy tipado. Una variable que contiene una cadena de caracteres al inicio del programa puede contener una matriz durante la ejecución. Esto ocasiona muchos errores, ya que el desarrollador nunca puede estar seguro del tipo de sus variables.

Ejemplo, en una función, sumar números puede ser desastroso si, al final, uno de los números esperados es una cadena de caracteres.

Creamos un nuevo proyecto "SegundoProyectoTs" en Visual Studio Code. Y seguidamente creamos el fichero errorDeTipado.ts con el siguiente código:

```
// Función
function suma(cifra01, cifra02) {
  return cifra01 + cifra02;
}
// Uso de la función suma
let primeraCifra = 1;
let segundaCifra = 3
let cadenaDeCaracteres = "abc";
console.log(suma(primeraCifra, segundaCifra)); // Muestra 4
console.log(suma(cadenaDeCaracteres, segundaCifra)); //Muestra abc3
```

Para transcompilar este fichero TypeScript, primero ejecutamos el comando:

```
tsc errorDeTipado.ts
```

generamos el fichero de extensión .js y lo ejecutamos directamente usando el comando:

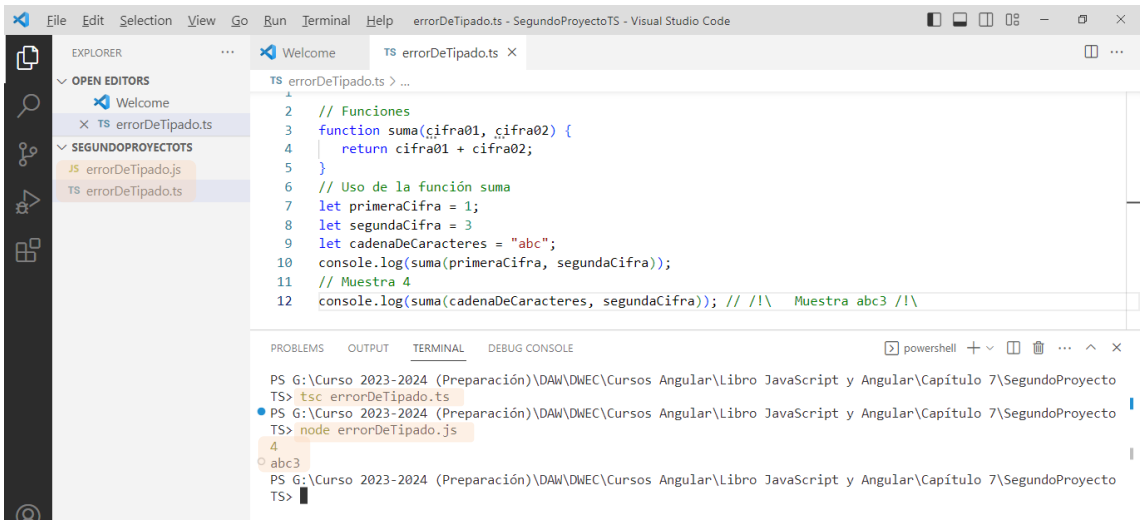
```
node errorDeTipado.js.
```

Debemos tener en cuenta, que si realizamos algún cambio en el fichero TypeScript, tendremos que ejecutar de nuevo el comando:

```
tsc errorDeTipado.ts
```

para volver a generar el fichero .js con dicho cambio y ejecutarlo posteriormente con el comando node:

```
node errorDeTipado.js
```



Como podemos observar en la imagen anterior, hemos usado el comando node para ejecutar el fichero TypeScript.

3.2 TypeScript y el tipado

TypeScript permite decirle al compilador y a los desarrolladores el tipo de dato de una variable e incluso el tipo de dato que devuelve una función.
Para hacer esto, agregamos a la declaración de la variable, su tipo precedido por **:**

```
let unaVariable : string;
```

Para una función, agregamos el tipo de retorno en la cabecera de la función, de la forma siguiente:

```
function unaFuncion() : string {
  return "una cadena de caracteres";
}
```

Ejemplo: Continuando con el ejemplo del apartado anterior, el código quedaría de la forma siguiente especificando el tipo de dato de las variables:
Creamos el fichero errorDeTipado2.ts, en el mismo proyecto, con el siguiente código:

```
// Función
function sumaConTipo(cifra01, cifra02) {
   return cifra01+ cifra02;
}
// Uso de la función sumaConTipo
let primeraCifraConTipo : number = 1;
let segundaCifraConTipo : number = 3
let cadenaDeCaracteresConTipo : number = "abc"; // Error en la transcompilación
```

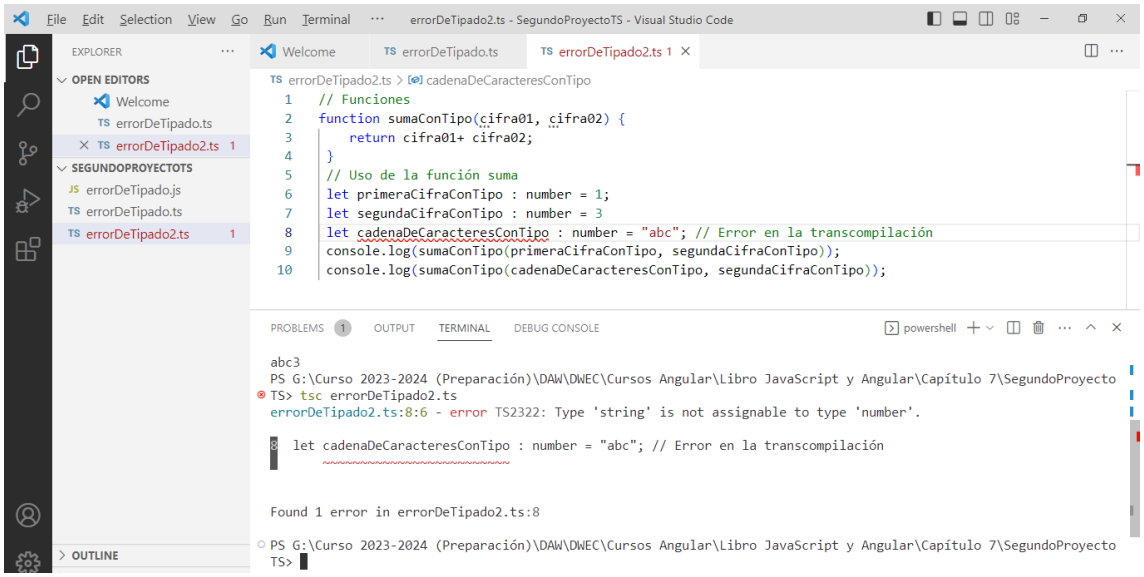

www.grupostudium.com
informacion@grupostudium.com
954 539 952

```
console.log(sumaConTipo(primeraCifraConTipo, segundaCifraConTipo));
console.log(sumaConTipo(cadenaDeCaracteresConTipo, segundaCifraConTipo));
```

Ejecutamos el comando:

```
tsc errorDeTipado2.ts
```

Para transcompilar el fichero TypeScritp y nos aparece el siguiente error que ya nos señalaba nuestro editor de código Visual Studio Code con la línea roja bajo la variable cadenaDeCaracteresConTipo.




```
PS G:\Curso 2023-2024 (Preparación)\DAW\DWEC\Cursos Angular\Libro JavaScript y Angular\Capítulo 7\SegundoProyecto
TS> tsc errorDeTipado2.ts
errorDeTipado2.ts:8:6 - error TS2322: Type 'string' is not assignable to type 'number'.

8 let cadenaDeCaracteresConTipo : number = "abc"; // Error en la transcompilación


Found 1 error in errorDeTipado2.ts:8
```

3.3 Errores de compilación

El siguiente código, perfectamente válido en JavaScript cuando el tipo de unaVariable no es definido en la declaración, en TypeScript informa de un error de compilación.

```
let unaVariable : string;
unaVariable = 17; // Error en la compilación
```

El entorno de desarrollo indica que el número 17 no se puede asignar a la variable unaVariable, que es de tipo string.

3.4 Tipos

En TypeScritp podemos usar 7 tipos de datos:

- Cadena de caracteres
- Número
- Booleano
- Array
- Enumeración
- Ningún tipo
- tipo vacío, que no está definido por el momento.

Veamos un ejemplo de cada uno de ellos:

```
// Cadena de caracteres
let unaCadenaDeCaracteres: string;
// Número
let unNumero: number;
// Booleano
let unBooleano: boolean;
// Array
let unArrayDeCadenaDeCaracteres: string[];
let unArrayDeNumeros: number[];
// Enumeración
enum unaEnumeracion {};
let unaVariableDeTipoEnumeracion: unaEnumeracion;
// Sin ningún tipo
let unaVariableSinTipo: void;
// Sin definición
let unaVariableSinDefinicion: any;
```

4. Enumeraciones

4.1 Declaración

Una enumeración es una lista de constantes como podemos ver en el siguiente ejemplo:

```
enum puntosCardinales {
  N,
  S,
  E,
  O
}
```

4.2 Uso de una variable de tipo enumeración

Creamos el fichero enumaracion.ts, en el mismo proyecto, con el código siguiente:

```
enum puntosCardinales {
  N,
  S,
  E,
  O
}
// Declaración
let unaDireccion: puntosCardinales;
// Uso
unaDireccion = puntosCardinales.N; // unaDireccion = 0
unaDireccion = puntosCardinales.S; // unaDireccion = 1
unaDireccion = puntosCardinales.E; // unaDireccion = 2
unaDireccion = puntosCardinales.O; // unaDireccion = 3
```

La variable unaDireccion sucesivamente toma la dirección norte, sur, este y oeste.

5. Funciones

5.1 Generalidades sobre Funciones

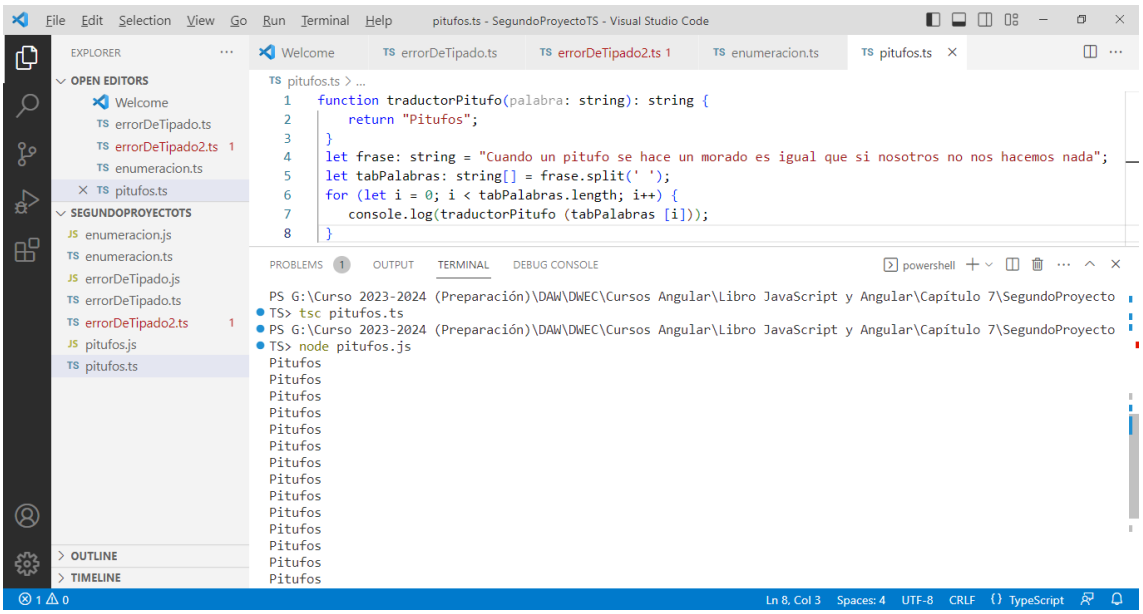
Al declarar una función usando TypeScript agrega algunos elementos a su cabecera, como vemos en el siguiente ejemplo:

```
function traductorPitufo(palabra: string): string {
  return "Pitufos";
}
let frase: string = "Cuando un pitufo se hace un morado es igual que si nosotros no nos hacemos nada";
let tabPalabras: string[] = frase.split(' ');
for (let i = 0; i < tabPalabras.length; i++) {
  console.log(traductorPitufo (tabPalabras [i]));
}
```

Creamos el fichero pitufos.ts en el mismo proyecto y lo ejecutamos usando los siguientes comandos como hemos visto en apartados anteriores:

```
tsc pitufos.ts
node pitufos.js
```

Observaremos la siguiente salida por consola:



Esta función toma una palabra de tipo cadena como argumento y la traduce al lenguaje Pitufo devolviendo una cadena.

5.2 Funciones particulares

5.2.1 Funciones anónimas

Una función podemos asignarla directamente a una variable, en ese caso hablamos de funciones anónimas.

Ejemplo:

```
let palabraTraducir = function traductorPitufo(palabra: string): string {
  return "Pitufos";
}
```

5.2.2 Funciones flecha

Las funciones flecha se han introducido desde ES6. El objetivo de las funciones flecha es el de simplificar más el uso de las funciones anónimas.

El ejemplo anterior de la variable palabraTraducir, podemos escribirlo de la forma siguiente usando las funciones flecha:

```
let palabraTraducir = palabras => "Pitufos";
```

Si la función tiene más de un argumento, se indicarán entre paréntesis. Respecto al cuerpo de la función, si contiene varias instrucciones, se pueden utilizar llaves.

Ejemplo:

```
let operacionCompleja = (cifra01, cifra02) => {
        let suma = cifra01+ cifra02;
        let resultado= suma * 20 + 2 - 123;
        return resultado;
}
```

6. Clases y herencia

6.1 Clase mínima

La sintaxis de una clase en TypeScript se parece mucho a la de una clase de Java.

Ejemplo:

```
// Definición de la clase coche
class Coche {
}
// Declaración de un objeto coche
let miCoche = new Coche();
```

Una clase en TypeScript tiene atributos, constructor y métodos y descriptores de acceso.

6.2 Atributos

Los atributos están precedidos por la palabra clave private para, encapsularlos, y que no puedan ser accedidos desde fuera de la clase. Se puede ver o modificar atributos gracias a los métodos get y set de cada atributo.

Ejemplo:

```
// Definición de la clase Coche
class Coche {
  // Atributos
  private _kilometraje: number;
  private _propietario: string;
  private _color: string;
}
// Declaración de un nuevo coche
let miCoche = new Coche();
```

6.3 Accesores (métodos getter y setter)

Los accesores nos permiten acceder y modificar los atributos privados de una clase. En la programación orientada a objetos, es esencial ocultar los atributos sensibles y convertir cada clase en una caja negra. Por lo tanto, encapsular es hacer que los atributos sean privados y ofrecer accesos controlados para acceder a ellos.

Ejemplo:

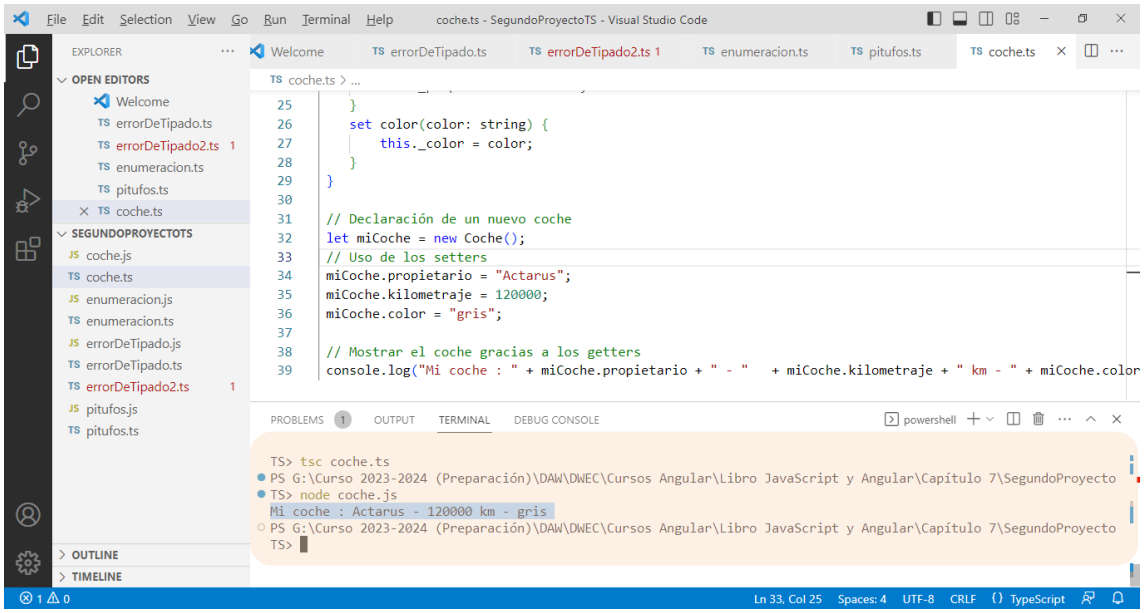
Creamos un nuevo fichero coche.ts con el siguiente código.

```
// Definición de la clase Coche
```

```
class Coche {
  // Atributos
  private _kilometraje: number;
  private _propietario: string;
  private _color: string;

  // Getters
  get kilometraje() {
     return this._kilometraje;
  }
  get propietario() {
     return this._propietario;
  }
  get color() {
     return this._color;
  }

  // Setters
  set kilometraje(km: number) {
     this._kilometraje = km;
  }
  set propietario(nombre: string) {
     this._propietario = nombre;
  }
 set color(color: string) {
     this._color = color;
 }
}

// Declaración de un nuevo coche
let miCoche = new Coche();
// Uso de los setters
miCoche.propietario = "Actarus";
miCoche.kilometraje = 120000;
miCoche.color = "gris";

// Mostrar el coche gracias a los getters
console.log("Mi coche : " + miCoche.propietario + " - "   + miCoche.kilometraje + " km
- " + miCoche.color);
```


www.grupostudium.com
informacion@grupostudium.com
954 539 952

Ejecutamos el fichero coche.js y observamos la salida por consola:



6.4 Constructor

Un constructor es una función un poco especial, ya que se ejecuta cuando se crea una nueva instancia del objeto. Sirve como función de inicialización. Su propósito es construir la instancia usando los parámetros dados por el desarrollador durante la declaración.

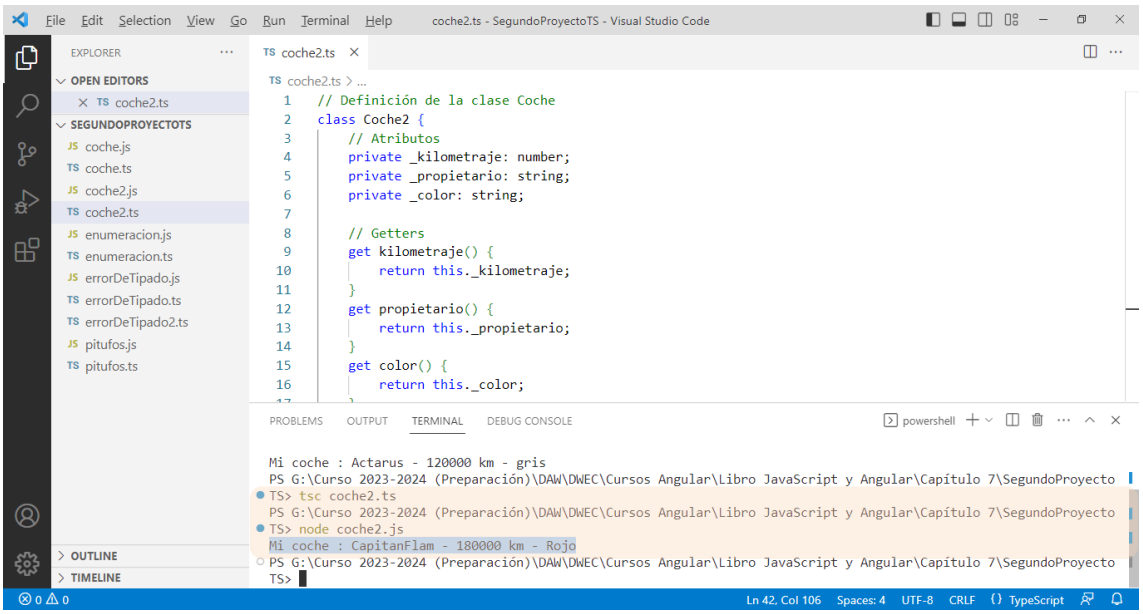
En ausencia de un constructor definido por el desarrollador, el compilador crea automáticamente un constructor vacío.

Ejemplo:

```
// Definición de la clase Coche
class Coche {
  // Atributos
  private _kilometraje: number;
  private _propietario: string;
  private _color: string;

  // Getters
  get kilometraje() {
     return this._kilometraje;
  }
  get propietario() {
     return this._propietario;
  }
  get color() {
     return this._color;
```

```
  }

  // Setters
  set kilometraje(km: number) {
     this._kilometraje = km;
  }
  set propietario(nombre: string) {
     this._propietario = nombre;
  }
     set color(color: string) {
     this._color = color;
  }

  // El constructor
  constructor(kilometraje: number, propietario: string, color: string) {
     this._kilometraje = kilometraje;
     this._propietario = propietario;
     this._color = color;
  }
}

// Declaración de un nuevo coche
let miCoche = new Coche(180000, "CapitanFlam", "Rojo");

// Mostrar el coche gracias a los getters
 console.log("Mi coche : " + miCoche.propietario + " - "   + miCoche.kilometraje + "
km - " + miCoche.color);
```

Ejecutamos el nuevo fichero coche2.ts y observamos la salida por consola:



6.5 Métodos

Un método es una función. Representa una posible acción del objeto.
Por ejemplo, un coche puede avanzar o retroceder.

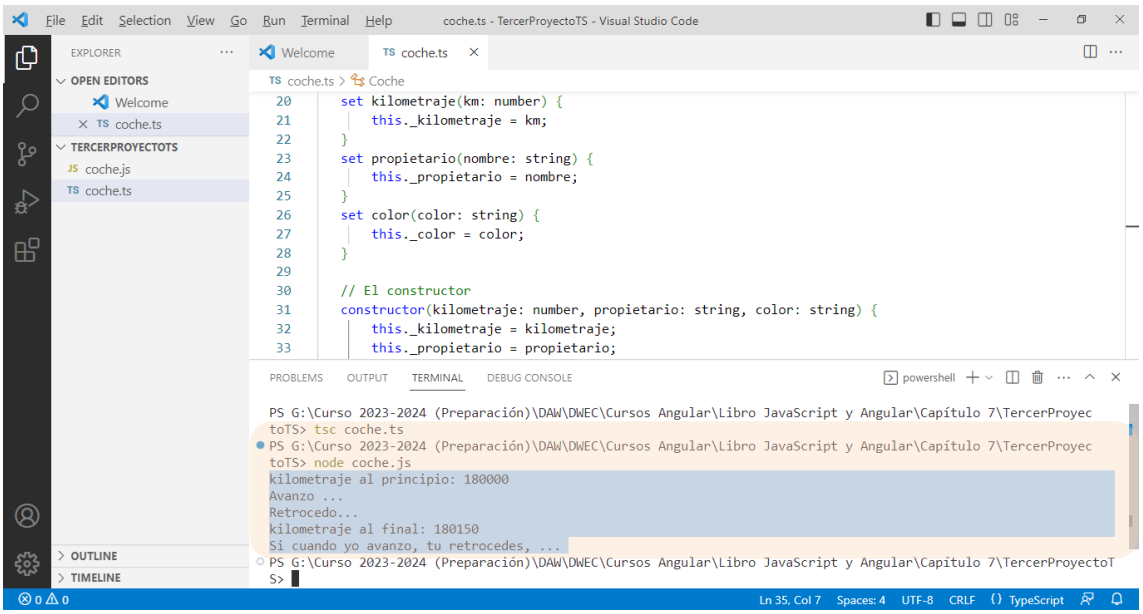
Ejemplo:

```
// Definición de la clase Coche
class Coche {
  // Atributos
  private _kilometraje: number;
  private _ propietario: string;
  private _color: string;

  // Getters
  get kilometraje() {
     return this._kilometraje;
}
  get propietario() {
     return this._propietario;
  }
  get color() {
     return this._color;
  }

  // Setters
  set kilometraje(km: number) {
     this._kilometraje = km;
```


www.grupostudium.com
informacion@grupostudium.com
954 539 952

```
  }
  set propietario(nombre: string) {
     this._propietario = nombre;
  }
  set color(color: string) {
     this._color = color;
  }

  // El constructor
  constructor(kilometraje: number, propietario: string, color: string) {
     this._kilometraje = kilometraje;
     this._propietario = propietario;
     this._color = color;
  }

  // Los métodos
  avanzar(numeroDeMetros: number) {
     this._kilometraje += numeroDeMetros;
  }
  retroceder(numeroDeMetros: number) {
      this._kilometraje += numeroDeMetros;  //Aunque el coche retroceda, aumenta
su kilometraje
  }
}

// Declaración de un nuevo coche
let miCoche = new Coche(180000, "CapitanFlam", "Rojo");
console.log("kilometraje al principio: " + miCoche.kilometraje);
console.log("Avanzo ...");
miCoche.avanzar(100);
console.log("Retrocedo...");
miCoche.retroceder (50);
console.log("kilometraje al final: " + miCoche.kilometraje);
console.log("Si cuando yo avanzo, tu retrocedes, ...")
```

Creamos un nuevo proyecto en Visual Studio Code, llamado TercerProyectoTS y dentro de él creamos el fichero coche.ts con el código indicado antes y lo ejecutamos para ver su salida por consola:



Con este ejemplo hemos visto el uso de una clase TypeScript completa.

6.6 Herencia

Una clase puede heredar de otra clase. La clase que hereda de la otra (subclase o clase hija) tiene los mismos atributos y métodos que esta (superclase o clase padre), pero también puede tener atributos propios.

Ejemplo:

Creamos la clase Vehiculo, que es idéntica a la clase Coche del ejercicio anterior, excepto por su nombre.

```
// Definición de la clase Vehículo
class Vehiculo {
  // Atributos
  private _kilometraje: number;
  private _propietario: string;
  private _color: string;

  // Getters
  get kilometraje() {
     return this._kilometraje;
  }
  get propietario() {
     return this._propietario;
  }
  get color() {
     return this._color;
  }
```


www.grupostudium.com
informacion@grupostudium.com
954 539 952

```
  // Setters
  set kilometraje(km: number) {
    this._kilometraje = km;
  }
  set propietario(nombre: string) {
    this._propietario = nombre;
  }
  set color(color: string) {
    this._color = color;
  }

  // El constructor
  constructor(kilometraje: number, propietario: string, color: string) {
    this._kilometraje = kilometraje;
    this._propietario = propietario;
    this._color = color;
  }

  // Los métodos
  avanzar(numeroDeMetros: number) {
    this._kilometraje += numeroDeMetros;
  }
  retroceder(numeroDeMetros: number) {
    this._kilometraje += numeroDeMetros;
  }
}
```

La clase CocheHeredado va a heredar la clase Vehiculo. No tiene atributos ni métodos adicionales. Y como hereda la clase Vehiculo, puede usar sus métodos y constructor:

```
// Definición de la clase CocheHeredado
class CocheHeredado  extends Vehiculo {

}

let tesla = new CocheHeredado  (666, "García Julián", "Azul Marino");
```

Creamos la clase TractorCortacesped que también hereda la clase Vehiculo, pero tiene dos métodos más:

```
class TractorCortacesped extends Vehiculo {
```

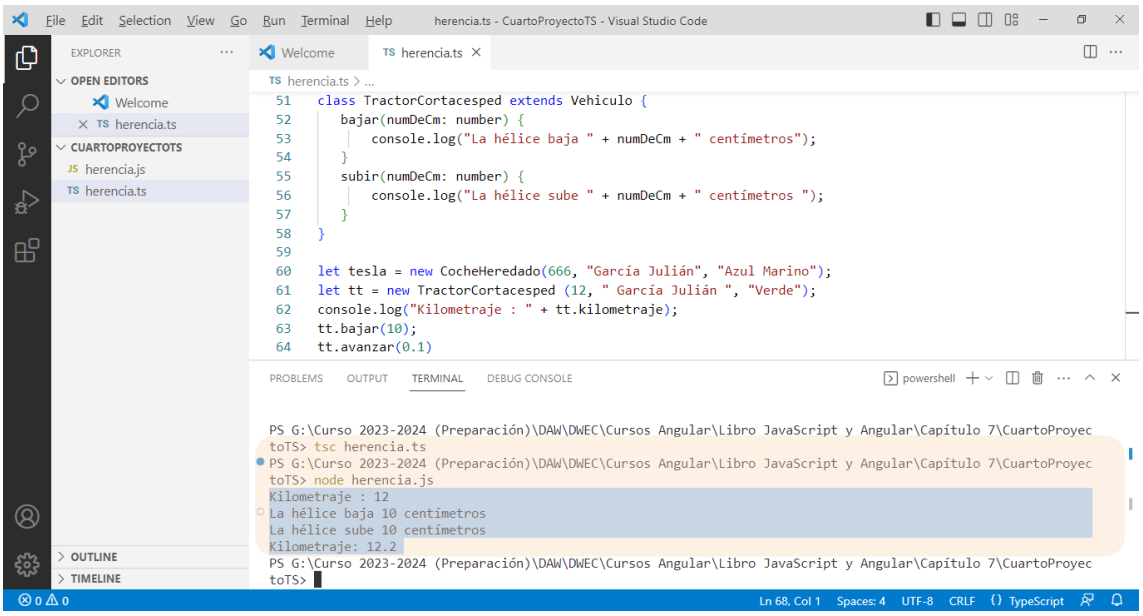
```
  bajar(numDeCm: number) {
     console.log("La hélice baja " + numDeCm + " centímetros");
  }
  subir(numDeCm: number) {
     console.log("La hélice sube " + numDeCm + " centímetros ");
  }
}

let tesla = new CocheHeredado(666, "García Julián", "Azul Marino");
let tt = new TractorCortacesped (12, " García Julián ", "Verde");
console.log("Kilometraje : " + tt.kilometraje);
tt.bajar(10);
tt.avanzar(0.1)
tt.subir(10);
tt.retroceder(0.1);
console.log("Kilometraje: " + tt.kilometraje);
```

Creamos el fichero herencia.ts completo en un nuevo proyecto y lo ejecutamos para ver su salida por consola:

```
// Definición de la clase Vehículo
class Vehiculo {
  // Atributos
  private _kilometraje: number;
  private _propietario: string;
  private _color: string;

  // Getters
  get kilometraje() {
     return this._kilometraje;
  }
  get propietario() {
     return this._propietario;
  }
  get color() {
     return this._color;
  }
   // Setters
  set kilometraje(km: number) {
     this._kilometraje = km;
  }
```

```
  set propietario(nombre: string) {
    this._propietario = nombre;
  }
  set color(color: string) {
    this._color = color;
  }

  // El constructor
  constructor(kilometraje: number, propietario: string, color: string) {
    this._kilometraje = kilometraje;
    this._propietario = propietario;
    this._color = color;
  }

  // Los métodos
  avanzar(numeroDeMetros: number) {
    this._kilometraje += numeroDeMetros;
  }
  retroceder(numeroDeMetros: number) {
    this._kilometraje += numeroDeMetros;
  }
}

// Definición de la clase CocheHeredado
class CocheHeredado  extends Vehiculo {

}

//Clase TractorCortacesped
class TractorCortacesped extends Vehiculo {
  bajar(numDeCm: number) {
    console.log("La hélice baja " + numDeCm + " centímetros");
  }
  subir(numDeCm: number) {
    console.log("La hélice sube " + numDeCm + " centímetros ");
  }
}

let tesla = new CocheHeredado(666, "García Julián", "Azul Marino");
let tt = new TractorCortacesped (12, " García Julián ", "Verde");
```

```
console.log("Kilometraje : " + tt.kilometraje);
tt.bajar(10);
tt.avanzar(0.1)
tt.subir(10);
tt.retroceder(0.1);
console.log("Kilometraje: " + tt.kilometraje);
```



Para construir un constructor en una clase que hereda de otra, es preferible usar el constructor de la clase que está heredando a la que agrega atributos adicionales. El método super() sirve precisamente para eso. Llama al constructor de la clase que estamos heredando.

7. Módulos

En los ejemplos anteriores, hemos podido observar que una clase, un atributo, una instancia... no se pueden declarar dos veces, incluso si están en dos archivos TypeScript diferentes. El motivo es que cada elemento declarado en TypeScript tiene un alcance global.

Esto puede plantear problemas en proyectos de gran tamaño y por ello apareció el concepto de módulo.

Un módulo es un conjunto de clases. Cada elemento del módulo tiene un alcance limitado al módulo, a menos que se exporte explícitamente. Lo que significa que muy bien podemos tener una clase Coche en un módulo miModulo01 y otra clase Coche en el módulo miModulo02.

Crear un módulo es simplemente insertar clases en una nueva declaración module.

www.grupostudium.com
informacion@grupostudium.com
954 539 952

Ejemplo:

```
module miModulo {

  class Vehiculo {
     metodoDelVehiculo() {
        console.log("Método del vehiculo.");
     }
  }

  class Coche extends Vehiculo {
     metodoDelCoche() {
        console.log("Método del coche.");
     }
  }

  class TractorCortacesped extends Vehiculo {
     metodoDelTractorCortacesped () {
        console.log("Método del tractor cortacésped.");
     }
  }
}
```

Para utilizar las clases y los métodos de un módulo, deben exportarse.

Ejemplo:

```
module miModulo {

  export class Vehiculo {

     constructor(marca: string, modelo: string) {
     }

     metodoDelVehiculo() {
        console.log("Método del vehiculo.");
     }
  }

  export class Coche extends Vehiculo {

     metodoDelCoche() {
```


www.grupostudium.com
informacion@grupostudium.com
954 539 952

```
        console.log("Método del coche.");
    }
  }

  export class TractorCortacesped extends Vehiculo {
    metodoDelTractorCortacesped () {
        console.log("Método del tractor cortacésped.");
    }
  }
}
let miCoche: miModulo.Vehiculo = new miModulo.Vehiculo("Tesla", "3");
miCoche.metodoDelVehiculo();
```

Creamos un nuevo proyecto y el fichero modulo.ts. Lo ejecutamos par ver su salida por consola: