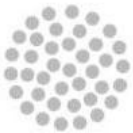
ESCUELA:
TECNOLOGÍA

TypeScript

12 al 15 de Noviembre de 2018



indra

OBJETIVOS

- Dominar TypeScript, superset de Javascript, el lenguaje que compila a Javascript compatible con todos los navegadores
- Enfocado al desarrollo de aplicaciones empresariales escalables y de alta calidad del código
- Orientar y facilitar el aprendizaje de Angular

ÍNDICE

ÍNDICE

- Definiendo y usando Interfaces
- Interfaces para Function types
- Extendiendo Interfaces
- Implementando interfaces con Clases
- Creando y usando Clases
- Extendiendo Clases, Creando Clases abstractas
- Namespaces, Modules, Decorators,
- Usando Expresiones Clase
- Opciones de Compilación con tsconfig

Herramientas de desarrollo

- Git for Windows
- Servidor JavaScript Node.js
  - Gestor de paquetes npm, incluido en Node.js
  - Instalador de paquetes yarn
- Editores
  - Atom editor
  - Visual Studio Code
- Entornos visuales de desarrollo
  - Plugin TypeScript para Eclipse
  - Visual Studio Community Edition

Instalación y configuración

- Instalar un cliente para Git
  - Configurar el cliente si estamos usando un proxy
- Instalar Node.js
  - Configurar el gestor de paquetes npm si estamos usando un proxy
- Instalar el gestor de paquetes yarn (opcional)
  - Configurar el gestor de paquetes yarn si estamos usando un proxy
- Instalar typescript: npm install -g typescript
- Instalar un servidor web para desarrollo: npm install -g lite-server
- Instrucciones para la configuración mínima de las herramientas
- Instalar el Visual Studio Code

Introducción a TypeScript

- TypeScript (a partir de ahora ts) es un superconjunto de JavaScript
- La herramienta tsc (TypeScript Compiler) traduce a JavaScript estándar (es3, es5, es6) las extensiones del lenguaje
- La forma más simple de trabajar con TypeScript es instalar Node.js, y mediante el gestor de paquetes npm:
  - npm install -g typescript
- Podemos experimentar desde el playground sin instalar nada en absoluto
- Los ejemplos oficiales de typescript muestran numerosos casos de uso
- En un proyecto real, lo normal es que se configure el tsc mediante el archivo **tsconfig.json**. Para compilar un archivo ts: tsc archivo.ts
- Para ejecutarlo: node archivo.js

Introducción a TypeScript (características)

- Tipos de datos
- Disponibilidad de var, let y const. Ver esta discusión
- Interfaces
- Clases
- Funciones
- Genéricos
- Tipos Enumerados

Introducción a TypeScript (características)

- Inferencia de tipos
- Compatibilidad estructural de tipos
- Símbolos
- Tipos intersección y unión
- Iteradores
- Módulos
- Espacios de nombres
- Módulos y espacios de nombres

Tipos básicos

- Boolean:
  const uno: boolean = true;
- Number:
  const uno: number = 25;
- String: const uno: string = ‘abc’;
  - Interpolación de Strings, empleando backticks: `El valor de uno es ${uno}`;
- Array:
  const uno: number[] = [1,2,3];
  const uno: Array<number> = [1,2,3];
- Tuple: agregados de valores, posiblemente heterogéneos, cuyo número y tipos son conocidos en tiempo de compilación
  const uno:[string,number] = ['abc', 25];
  const uno:[string,number, boolean] = ['abc', 25, true];

Tipos básicos

- Enum

```
enum Lenguajes { TypeScript, Java, Kotlin };
const r = Lenguajes.TypeScript === 0; // r es true
```

- Any: para facilitar la integración con JS. Representa cualquier tipo

```
let uno: any = 8;
uno = ‘Legal, porque el tipo es any’;
```

- Void: ausencia de cualquier tipo

```
function efectoColateral(mensaje: string): void {
    console.log(`El mensaje es ${mensaje}`);
}
efectoColateral('abc');
```

- Undefined y Null: subtipo de todos los demás tipos

```
let uno: undefined = undefined;
let uno: null = null;
```

Tipos básicos

- Never: representa tipos que nunca ocurren. Subtipo de todos los demás

```
function lanzarError(mensaje: string): never{
    throw new Error(mensaje);
}
```

- Object: todo lo que no sea un tipo de dato “primitivo”

```
function toString(obj: object): string{
    const r:string[] = [];
    for (const n in obj) {
        const valor = `obj.${n} = ${obj[n]}`;
        r.push(valor);
        console.log(valor);
    }
    return r.join(',');
}
```

- Conversiones forzadas (Type assertions)

```
const uno: any = 'abc';
const dos = uno as string;
```

Declaraciones de variables

- Podemos usar var, let y const
- Var pertenece a JavaScript. Preferimos no usarlo en TS para evitar una serie de problemas asociados a los espacios de nombres globales estándar en JS
- Ejecutar:

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

  - y observar el resultado. Cambiar var por “let o const”, ejecutar y observar el resultado
- Usaremos “let” para declarar variables
- Si a una variable queremos asignarle un valor y no cambiarlo, preferimos emplear “const”. Aunque declaremos un objeto como constante, sigue siendo posible modificar sus propiedades

Desestructuración y extensión de objetos

- Desestructurar objetos es útil para descomponer automáticamente un agregado complejo en variables más simples

```
const arr = [1, 2, 3];
let [uno, dos, tres] = arr;
console.log([tres, dos, uno]);
document.body.innerHTML = [tres, dos, uno].toString();
```

- Otros ejemplos:

```
const arr = Array.from(Array(5).keys());
let [uno, ...dos] = arr;
console.log([dos,uno]);
document.body.innerHTML = [dos,uno].toString();

function f([uno, dos]: [number, number]) {
    return uno + dos;
}
document.body.innerHTML = f([10, 10]).toString();
```

Desestructuración y extensión de objetos

- La desestructuración puede también aplicarse a objetos

```
const objeto = {
    str: 'abc',
    numero: 12,
    funcion: (x) => x * 2
};
let { str, numero, funcion } = objeto;
document.body.innerHTML = [numero,str,funcion(10)].toString();
```

- La extensión es opuesta a la desestructuración:

```
const arr = [1, 2, 3];
const v = [10,...arr.reverse(),100];
document.body.innerHTML = v.toString();
```

- También puede aplicarse con objetos:

```
let props = { str: 'abc', numero:20, fecha: new Date() };
let resultado = { ...props, str: 'Valor sobreescrito' };
document.body.innerHTML = JSON.stringify(resultado);
```

Interfaces

- En TS, dos objetos son compatibles si comparten la misma estructura:

```
function f(objeto: { nombre: string }): string{
    return objeto.nombre.toUpperCase();
}
const uno = { nombre: 'abc' };
const nodo = document.createElement("div");
nodo.innerHTML = f(uno);
document.body.appendChild(nodo);
const dos = { fecha: new Date(), nombre: 'def' };
const nodo1 = document.createElement("div");
nodo1.innerHTML = f(dos);
document.body.appendChild(nodo1);
```

- Desde el punto de vista de TS, los objetos “uno” y “dos” son parámetros legales de la función “f”. Ambos comparten la estructura que dicha función requiere.

Interfaces

- En TS, una interface da un nombre a una definición de tipo

```
interface Nombrable{ nombre: string; }
function f(objeto: Nombrable): string{
    return objeto.nombre.toUpperCase();
}
const uno = { fecha: new Date(), nombre: 'abc' };
const nodo = document.createElement("div");
nodo.innerHTML = f(uno);
document.body.appendChild(nodo);
```

- Una interfaz puede tener propiedades opcionales:

```
interface Persona{
    nombre: string;
    edad?: number;
}
const p1: Persona = { nombre: 'abc', edad: 25 };
const p2: Persona = { nombre: 'def' };
```

Interfaces

- Una interfaz puede tener propiedades de sólo lectura, el equivalente a usar “const” en la declaración de objetos:

```
interface Persona{
    readonly nombre: string;
    readonly edad: number;
}
const p1: Persona = { nombre: 'abc', edad: 25 };
p1.edad = 30; //Error
```

- Véase cómo creamos objetos conforme a una interfaz. Simplemente empleamos una de las sintaxis estándar de JS para declarar un objeto.

Interfaces

- Usando interfaces no sólo podemos describir objetos, sino también funciones:

```
interface FuncionDef{
    (x: number, y: number): number;
}
const suma: FuncionDef = function (x, y) { return x + y; }
const mult: FuncionDef = function (x, y) { return x * y; }
function usarFuncionDef(x: number, y: number, f: FuncionDef) {
    return `El resultado es: ${f(x, y)}`;
}
document.body.innerHTML = usarFuncionDef(10, 10, mult);
```

- También es posible representar tipos indexables:

```
interface IndexDef{    [indice: number]: string;}
const arr: IndexDef = ['1', '2', '3'];
document.body.innerHTML = arr[0];
```

Interfaces

- Una interfaz puede extender a una o varias interfaces, esto es, adquirir su estructura

```
interface Uno{
    prop1: string;
}
interface Dos extends Uno{
    prop2: number;
}
interface Tres extends Uno, Dos{
    prop3: Date;
}
const dos: Dos = {} as Dos; // Las propiedades no están definidas
const tres: Tres = <Tres>{}; // Las propiedades no están definidas
```

- Cuando una clase quiere ser conforme a una interfaz debe implementarla:

```
class A implements Uno { } // Error
class A implements Uno {  prop1: string; }
```

Clases

- TS permite programar empleando la orientación a objetos clásica: interfaces, clases, jerarquías de herencia entre clases, polimorfismo, ámbitos de visibilidad, etc
- Declaración de una clase: class A{}. Creación de un objeto: const a = new A();
- Las clases pueden tener atributos (propiedades), métodos y un constructor. El ámbito de visibilidad por defecto es público (public)

```
class A{
    prop: string;
    mostrarProp() { console.log(this.prop);  }
    constructor(prop: string) {
        this.prop = prop.toUpperCase();
    }
}
const a = new A('abc');
document.body.innerHTML = a.prop;
```

Clases

- Jerarquías de herencia:

```
class A{
    constructor(public nombre: string) {}
}
class B extends A{
    constructor(public numero: number, nombre: string) {
        super(nombre);
    }
}
const a = new B(25,'abc');
console.log(a.nombre + ' ' + a.numero);
document.body.innerHTML = a.nombre + ' ' + a.numero;
```

Clases

- Redefinición de métodos en descendientes:

```
class A{
    mostrarProp() { console.log(this.nombre); }
    constructor(public nombre: string) {}
}
class B extends A{
    constructor(public numero: number, nombre: string) {
        super(nombre);
    }
    mostrarProp() {
        super.mostrarProp();
        console.log(this.numero;
    }
}
const a = new B(25,'abc');
a.mostrarProp();
document.body.innerHTML = a.nombre + ' ' + a.numero;
```

Clases

- Ámbitos de visibilidad:
  - public, globalmente accesible
  - protected, accesible en la clase en la que declaramos y en sus descendientes
  - private, accesible en la clase en la que declaramos
- En un constructor con parámetros públicos, privados o protegidos, las propiedades se generan automáticamente
- Las propiedades pueden ser de sólo lectura. Deben inicializarse al declararlas o en el constructor:

```
class A{
    private readonly direcciones = ['una','dos'];
    constructor(public nombre: string, public readonly fecha: Date) {}
}
class B extends A{
    constructor(public numero: number, nombre: string) {
        super(nombre, new Date());
    }
}
```

Clases

- Selectores y modificadores (getters y setters):

```
class A{
    private _nombre: string;

    get nombre() {
        console.log("Get de la propiedad nombre");
        return this._nombre;
    }
    set nombre(nuevo: string) {
        console.log("Set de la propiedad nombre");
        this._nombre = nuevo;
    }
}
const a = new A();
a.nombre = 'nuevo';
const v = a.nombre;
```

Clases

- Propiedades y métodos estáticos:

```
class A{
    static serialId = 1000;
    static metodo() { return 0;}
    constructor(public nombre: string) { }
}
A.serialId = 2000; const v = A.metodo();
new A('abc').metodo(); //Error
```

- Clases y métodos abstractos:

```
abstract class A{
    constructor(public nombre: string) { }
    abstract metodo(): number;
}
class B extends A{
    constructor(public n: number, s: string) { super(s); }
    metodo() { return 0;}
}
const a = new A('abc'); //Error
```

Clases

- En TS, las clases pueden emplearse como interfaces:

```
class A{
    n: string;
}
interface B extends A{
    m: number;
}
const v = { n: 'abc', m: 25 } as B;
```

Clases

- Como en la OO clásica, una clase puede implementar 0..N interfaces y participar en jerarquías de herencia al hacerlo:

```
interface A{
    metodoA(): string;
}
interface B{
    metodoB(): number;
}
class C implements A, B{
    metodoA() { return ''; }
    metodoB() { return 0;}
}
class D{ }
class F extends D implements A, B{
    metodoA() { return ''; }
    metodoB() { return 0;}
}
```

Funciones

- Además de los métodos definidos en clases, TS soporta completamente el uso de funciones
- Con la sintaxis estándar de JS: function f(a,b){ return a + b; }
- Añadiendo información de tipo (Function Types):

```
function f(a:number, b: number) : number{ return a + b; }
```

- Funciones anónimas:

```
const f = function (a:number, b: number) : number{ return a + b; }
```

- El tipo de una función puede escribirse explícitamente:

```
const f: (uno:number, dos:number) => number =
function (a:number, b: number) : number{ return a + b; }
```

- Si no escribimos el tipo de la función, TS tratará de inferirlo automáticamente
- Podemos declarar parámetros opcionales:

```
function f(a:number, b?:number):number{
if(b) return a + b;
else return a;
}
```

Funciones

- Los parámetros de una función pueden tener valores por defecto:

```
function f(a: number, b: number = 1): number{
    return a + b;
}
```

- Observar el tipo inferido por TS: el segundo parámetro se trata como opcional
- Una función puede tener una lista de parámetros específica y el “resto”:

```
function f(a: number, b: number = 1, ...resto: number[]): number{
    return resto.reduce((x, y) => x + y,a+b);
}
console.log(f(1, 1, 2, 2) === 6) //true
console.log(f(1) === 2) //true
```

Funciones flecha

- Una función “flecha” pretende ser una abreviatura en relación con la sintaxis estándar de funciones:

```
function suma(a: number, b: number): number{ return a + b; }
const f: (a: number, b: number) => number = (a, b) => a + b;
```

- Sin embargo, tal vez el aporte más importante sea el “this contextual”:

```
const objeto = {
    metodo: function f() {
        return function () {
            console.log(this);
        }
    }
}
objeto.metodo()()
```

- Si ejecutamos esta secuencia, veremos que this no apunta a la constante objeto, sino a Window.

Funciones flecha

- En JS (y TS) el valor de this al cual se asocia un objeto (función) se adquiere cuando el objeto (función) se ejecuta, no cuando se declara
- En cambio, al usar funciones flecha:

```
const objeto = {
    str: 'abc',
    metodo: function f() {
        return () => console.log(this);
    }
}
objeto.metodo()()
```

- Veremos que this apunta al objeto en el cual la función se declaró
- En otras palabras, al usar funciones flecha, this siempre apunta donde esperamos que apunte

Funciones flecha

- Pueden usarse para definir métodos:

```
class FuncionesFlecha{
    constructor(public str: string) { }
    metodo: () => string = () => this.str.toUpperCase();
}
const f = new FuncionesFlecha('abc');
console.log(f.metodo()); // ABC
```

- Si una función flecha devuelve objetos:

```
const f = () => { numero: 10; }
console.log(f()); //undefined
```

- Hay que corregir la sintaxis un poco:

```
const f1 = () => ({ numero: 10; })
console.log(f1());
```

- Observar en los dos ejemplos anteriores el tipo de la constante inferido por TS

Arrays

- Conjuntos de objetos que pueden ser tratados como secuencias o como mapas

```
const arr = [1, 2, 3];
arr[0] = 1000;
console.log(arr[0]);
```

- También es posible tratar un array como un mapa (conjunto de pares clave, valor)

```
const arr = [1, 2, 3];
arr['test'] = 'test';
console.log(arr["test"]);
arr[1000] = 25;
for (const x in arr) {
    console.log(`Clave = ${x}. Valor = ${arr[x]}`);
}
```

Arrays

- JS siempre ha sido un lenguaje funcional y, por tanto, TS también
- Los arrays han adquirido gradualmente una serie de métodos que implementan operaciones de transformación funcional típicas
- Transformación (map):

```
const r1 = Array.of(1, 2, 3);
const r2 = r1.map(n => n + 1);
console.log(r1);
console.log(r2);
```

- Selección (filter):

```
const r1 = Array.from('test de prueba');
const r2 = r1.filter(n => n === 'e');
console.log(r1);
console.log(r2);
```

- Reducción (reduce):

```
const r = [1, 2, 3];
console.log(r.reduce((a, b) => a + b));
```

Arrays

- Ordenación (sort):

```
const r = [3,2,1];
console.log(r.sort());
const r1 = ['tres','cuatro','uno'];
console.log(r1.sort((x,y) => x.length - y.length));
```

- Simplificación estructural (flat):

```
const r = [3,[2,4,5],1];
console.log(r.flat());
```

- Simplificación estructural y transformación (flatMap):

```
const r = ['uno', 'dos', 'tres'];
console.log(r.map(c => [c.toUpperCase()]));
console.log(r.flatMap(c => [c.toUpperCase()]));
```

- Iteración (forEach):

```
const r = ['uno', 'dos', 'tres'];
r.forEach(str => console.log(`El valor es ${str}`));
```

Genéricos

- El uso de genéricos facilita la reutilización de código, en el sentido de aplicar la misma lógica a cualquier objeto, independientemente de su tipo
- Por ejemplo:

```
function porConsola(a) {
    console.log(a);
}
```

- Si queremos especificar completamente el tipo de esta función en TS, podríamos escribir:

```
function porConsola(a: any): void {
    console.log(a);
}
```

- Al usar “any” perdemos toda la protección del compilador de TS.

Genéricos

- Para recuperarla, y no obstante poder aplicar la función a cualquier objeto:

```
function porConsola<T>(a: T): T {
    console.log(a);
    return a;
}
const num: number = porConsola(8);
const str: string = porConsola('8');
```

- Es posible emplear funciones flecha:

```
const porConsola: <T>(a: T) => T = a => {
    console.log(a);
    return a;
};
const num: number = porConsola(8);
const str: string = porConsola('8');
```

Genéricos

- Las clases pueden usar genéricos:

```
class Pila<T>{
    almacen:T[] = [];
    push(n:T) {
        this.almacen.push(n);
    }
    pop():T {
        const x = this.almacen[0];
        this.almacen = this.almacen.slice(1, this.almacen.length);
        return x;
    }
}
const pila1 = new Pila<string>();
pila1.push('abcv');
```

Genéricos

- Así como las interfaces:

```
interface Repositorio<T>{
    guardar(objeto: T): T;
    sufijo(texto: string): T[];
}
class RepoString implements Repositorio<string>{
    private cache: string[] = [];
    constructor(...valores: string[]) {
        valores.forEach(v => this.cache.push(v));
    }
    guardar(objeto: string): string{
        this.cache.push(objeto);
        return objeto;
    }
    sufijo(texto: string): string[]{
        return this.cache.map(s => s.concat(texto));
    }
}
const r = new RepoString('uno', 'dos');
r.guardar('tres');
console.log(r.sufijo('test'));
```

Expresiones Clase

- En TS (y JS) podríamos decir que una “Class Expression” es una clase anónima:

```
const Rectangulo = class {
  constructor(public alto, public ancho) {
  }
  area() {
    return this.alto * this.ancho;
  }
}
console.log(new Rectangulo(5,8).area());
```

Decoradores

- Un decorador suministra información adicional a TS mediante el uso combinado de la sintaxis @decorador y la función que le respalda

```
function log() {
    console.log('La función log ha sido evaluada');
    return function (target, propertyKey: string, descriptor:
PropertyDescriptor) {
        console.log('La función log ha sido llamada');
        console.log(target);
        console.log(propertyKey);
        console.log(descriptor);
    }
}
class Cualquiera {
    @log()  metodoLogeable() { console.log('Prueba'); }
}
const c = new Cualquiera();
c.metodoLogeable()
```

Decoradores

- Pueden aplicarse típicamente a clases, métodos, selectores y modificadores (get(), set()), propiedades y a parámetros
- En el momento de escribir esto, los decoradores son una funcionalidad experimental que por defecto no está activada y puede sufrir cambios importantes en el futuro
- En general, la función que respalda a un decorador debe retornar otra función
- Para experimentar con decoradores:

```
tsc --target ES5 --experimentalDecorators codigo.ts
node codigo.js
del codigo.js
```

- Ver ejemplo en typescript\ejemplos\decorators

Espacios de nombres

- Mecanismo para organizar nuestro código fuente

```
namespace Persistencia{
  export interface Identificable{
    id: number;
  }
  const repo: Identificable[] = [];
  export class Base implements Identificable{
    constructor(public id: number) { }
    guardar() {
      repo.push(this);
    }
    mostrar() {
      repo.filter(item => item.id === this.id).forEach(item =>
console.log(item));
    }
  }
}
```

Espacios de nombres

- La idea es similar a los packages de Java, por poner un ejemplo
- Una vez definidos los espacios de nombres y exportado lo que nos interesa:

```
class Persona extends Persistencia.Base{
  constructor(public nombre: string = "", public id: number = 0) {
    super(id);
  }
}
const p = new Persona('abc', 25);
p.guardar();
p.mostrar();
```

- Es posible emplear varios archivos de código fuente para definir y usar los espacios de nombres. Ver el ejemplo en typescript\ejemplos\namespaces

Módulos

- Un módulo representa otra manera de reorganizar nuestro código fuente
- Los módulos se ejecutan en su propio ámbito (scope) por lo que deseemos hacer visible a otros módulos debe ser exportado (export)
- Si el módulo necesita una serie de artefactos existentes en otros módulos deben ser importados (import)
- Ver un ejemplo en typescript\ejemplos\modules
- Exportar objetos, algunos ejemplos:

```
export const numero = 9;
export interface A{}
export class A{}
class A{}
export { A };
export { A as OtraClase }
```

Módulos

- Importar objetos, algunos ejemplos:

```
import { A } from ‘./clases’;
import { A as Otra } from ‘./clases’;
import *  as espacioNombres from ‘./clases’;
const a = new espacioNombres.A();
import ‘./globales.js’;
export default class A{} //módulo clases.ts
import DaIgual from ‘./clases’;
const a = new DaIgual(); //Crea una instancia de la clase A
```

- Se pueden exportar por defecto valores y funciones
- Ver las diferencias y convergencias entre namespaces y modules

Configuración y compilación

- Si en una carpeta tecleamos la orden: tsc --init se genera el archivo tsconfig.json
- En este archivo es donde decidimos cómo se comportará TS en función de las opciones de compilación que activemos o desactivemos
- Para TS, la presencia de este archivo indica que está tratando con un proyecto, no con archivos aislados
- Así, al invocar al compilador sin indicarle específicamente sobre què archivos deseamos que opere, TS buscará, leerá y “ejecutará” el archivo de configuración
- Por ejemplo
  - tsc + ENTER. TS compilará todos los archivos ts que encuentre
  - tsc --watch + ENTER. Lo mismo, pero el compilador detectará cambios en el código fuente y recompilará los archivos afectados

Integración con WebPack

- Crear el directorio “integracionWebPackUno”
  - mkdir integracionWebPackUno && cd integracionWebPackUno
- Crear un proyecto npm (package.json)
  - npm init -y. Inspeccionar el archivo “package.json”
- Crear el archivo “tsconfig.json”

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

Integración con WebPack

- Instalar Webpack y algunas dependencias
  - npm install webpack webpack-cli webpack-dev-server *--save-dev*
- Instalar localmente TypeScript y su cargador
  - npm install typescript ts-loader --save-dev
- Inspeccionar el archivo "package.json"
- Crear el archivo "app.ts"
  - Definir la clase Persona(nombre, edad) y crear un método toString para obtener los datos de una instancia
  - En "app.ts" introducir el código necesario para mostrar los datos de una persona bien por consola (console.log…), bien en la propia página o en ambas
- Crear el archivo "index.html" e introducir una referencia a "app.js" que crearemos con Webpack

Integración con WebPack

- Crear el archivo “webpack.config.js”

```
const path = require('path');
module.exports = {
    entry: path.join(__dirname, '/app.ts'),
    output: {
        filename: 'app.js',
        path: __dirname
    },
    module: {
        rules: [
            {
                test: /\.tsx?$/,
                loader: 'ts-loader',
                exclude: /node_modules/,
            },
        ]
    },
    resolve: {
        extensions: [".tsx", ".ts", ".js"]
    },
};
```

Integración con WebPack

- Modificar “package.json” y colocar dentro de la etiqueta “scripts” una nueva tarea, que npm reconoce automáticamente, llamada “start”
  - "start": "webpack-dev-server --mode development"
- Desde una ventana de comandos, escribir: npm start y pulsar ENTER
- En el navegador, ir a http://localhost:8080
- Aparecerá la página principal del ejemplo
- Es muy común emplear WebPack para crear “bundles”
- Ver typescript\integración\ejemploBundle