actibyti] a netmind social project

TYPESCRIPT

Autor: Ricardo Ahumada

GOBIERNO DE ESPAÑA
MINISTERIO DE ENERGÍA, TURISMO Y AGENDA DIGITAL
red.es
ESTRATEGIA DE EMPRENDIMIENTO Y EMPLEO JOVEN garantía juvenil
Agenda Digital para España
UNIÓN EUROPEA
Fondo Social Europeo
"El FSE invierte en tu futuro"

ÍNDICE DE CONTENIDOS

1. Introducción a TypeScript
2. Gramática, declaraciones y anotaciones
3. Clases e interfaces
4. Herencia
5. Definición de métodos, propiedades y campos
6. Módulos y la declaración de importación y exportación

2 TYPESCRIPT 2

TypeScript

- El lenguaje JavaScript nunca se pensó para realizar aplicaciones de gran envergadura.
- Por mantener la compatibilidad hacia atrás, se ha visto lastrado a la hora de establecer modelos más actuales de desarrollo
- **Brendan Eich** (su autor) realizó el primer prototipo en 12 días.
- A partir de la versión 5 (ECMAScript 262 5ª Edición), se planea la actualización a un modelo OOP declarando ciertas palabras "reservadas para una futura versión"

Ciclo de vida

- El ciclo de vida de un archivo TypeScript puede resumirse en el siguiente Gráfico

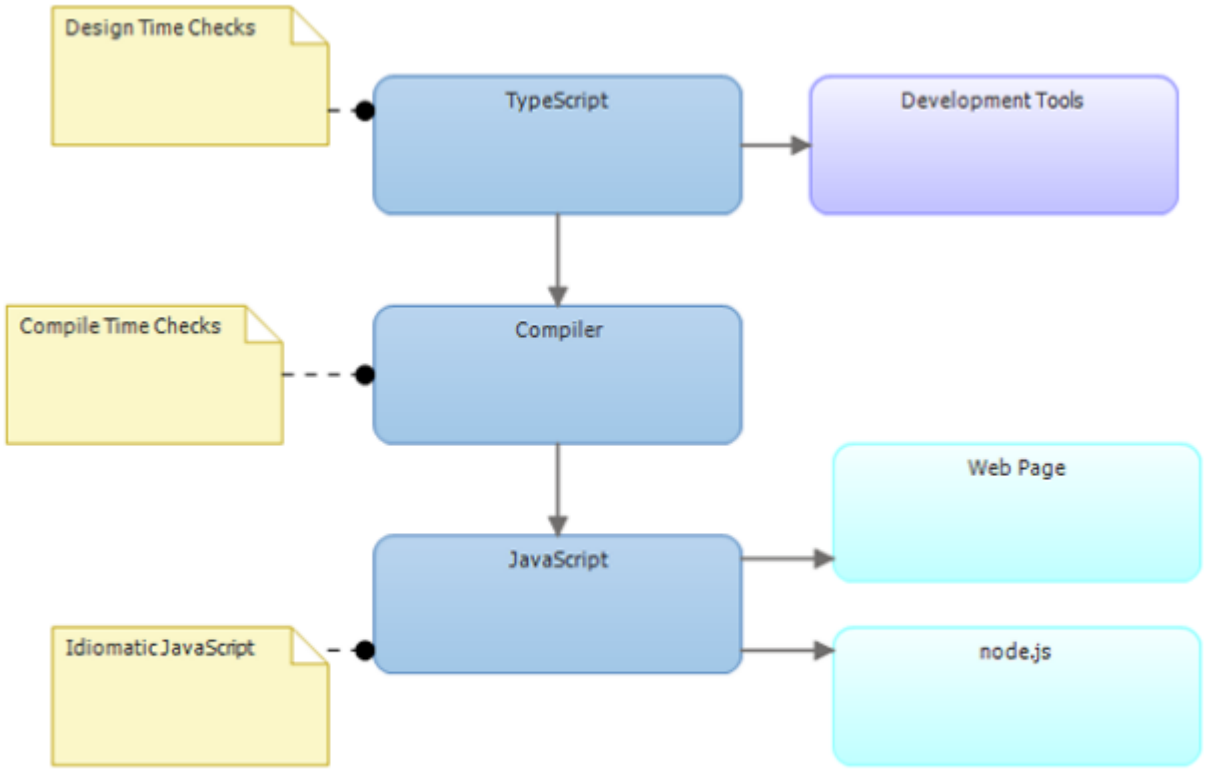

Figure 2: TypeScript life cycle

Herramienta PlayGround

- Es una página Web diseñada para editar y compilar código TypeScript en línea, que permite compartir código y ejecutar el JavaScript resultante. (http://www.typescriptlang.org/Playground/)

Declaraciones tipadas

- TypeScript permite la declaración de tipos en cualquier definición y reconoce el tipo correspondiente desde el IDE
- Decimos aquí que estamos usando una anotación de tipos
- Tipos básicos Typescript:
  - https://www.typescriptlang.org/docs/handbook/basic-types.html

```
function process(x: string) {
    var v = x + x;
    alert(v);
}
```

Declaraciones tipadas: Tipos complejos

- Los tipos complejos básicos (como *arrays*) se soportan de la misma forma (basta con indicar doble corchete al final:

```
function process4(x: string[]) {
}
```

Declaraciones tipadas: Tipos complejos

- Los interfaces se prestan especialmente bien para este tipo de declaraciones:

```
interface Monumento {
    nombre: string;
    alturaEnMetros: number;
}

// Aquí se usa la interfaz Monumento
var monumentos: Monumento[] = [];

// Cada elemento añadido se comprueba en compatibilidad
monumentos.push({
    nombre: 'Estatua de la Libertad',
    alturaEnMetros: 46,
    ubicacion: 'EE.UU.'
});
...
// Mas adelante podemos crear métodos de comparación
```

Comparador

```
monumentos.push({
    nombre: 'Pedro el Grande',
    alturaEnMetros: 96
});
monumentos.push({
    nombre: 'Angel del Norte',
    alturaEnMetros: 20
});

function compararAlturas(a: Monumento, b: Monumento) {
    if (a.alturaEnMetros > b.alturaEnMetros) {
        return -1;
    }
    if (a.alturaEnMetros < b.alturaEnMetros) {
        return 1;
    }
    return 0;
}

// El método array.sort espera un comparador que acepte 2 argumentos
var monumentosPorAltura = monumentos.sort(compararAlturas);
// Lee el primer elemento del array que es el mas alto.
var monumentoMasAlto = monumentosPorAltura[0];
console.log(monumentoMasAlto.nombre); // Pedro el Grande
```

Declaraciones tipadas: Expresiones *Lambda*

- Las declaraciones de funciones pasadas como argumento, se hacen mediante expresiones ***lambda*** (volveremos a esto más adelante)
- Se puede considerar que las expresiones lambda definen **contratos.**
- En este caso, el contrato indica que ***process5*** recibe un argumento de tipo ***function***, que no lleva parámetros, y devuelve una cadena.

```
function process5(x: () => string){
    x();
}
```

Declaraciones tipadas: **Objetos**

- Igualmente, podemos definir una función que recibe un objeto cualquiera, y dispondremos de "*Intellisense*"
- Aquí, ***process6*** recibe como argumento un ***object*** cuya estructura doble consta de un número y una cadena.

```
function process6( x: {a: number; b: string} ) {
  return x.a;
}
```

Algunos ejemplos de anotación de tipos

```
// array anotado
var nombres: string[] = ['Obdulio', 'Eufronio', 'Palmacio', 'Monglorio'];

// función  con parámetro y retorno anotados
var decirHola: (nombre: string) => string;

// implementacion de la función decirHola
// (para el compilador la anotación anterior funciona como si fuera una // // // interfaz)
decirHola = function (nombre: string) {
    return 'Hola, ' + nombre;
};

// objeto anotado
var persona: { nombre: string; alturaCM: number; };

// Implementacion del objeto persona
persona = {
    nombre: 'Monglorio',
    alturaCM: 183
};
```

Interfaces

- Al igual que sucede en lenguajes OOP (Java, C#), podemos declarar **interfaces**: tipos personalizados cuya estructura se define esos "contratos", que el contexto comprueba en tiempo de compilación.
- Esas interfaces permiten una declaración externa de contratos, reutilizable a través de la aplicación.

```
interface Cosa {
    a: number;
    b: string;
    c?: boolean; //opcional
}

var llamar = process7({ a: 10, b: "Saludo" });
```

Interfaces

- Así pues, mediante las interfaces se establecen las expectativas sobre el comportamiento de los objetos. Esto incluye miembros obligatorios u opcionales, funciones (métodos), eventos, etc.

- Por ejemplo, podemos declarar la interfaz Amigo de la siguiente forma:

```
interface Amigo {
        nombre: string;
        colorFavorito?: string;
}
```

- Si más adelante creamos una función que utilice esa interfaz en la declaración del argumento, la sintaxis será:

```
function nuevoAmigo(amigo: Amigo){
        var nombre = amigo.nombre;
}
```

Interfaces

- En el momento de utilizarla, el entorno reconocerá las llamadas incorrectas, indicándolo al subrayar la segunda de estas llamadas, que no incluye el parámetro obligatorio nombre:

```
nuevoAmigo({ nombre: "Paco" }); // Ok
// Error, se requiere el nombre
nuevoAmigo({ colorFavorito: "Amarillo" });
nuevoAmigo({ nombre: "Cristina", colorFavorito: "Rojo" }); // Ok
```

- Como veremos al utilizar las clases, esto permite crear constructores personalizados que disponen de "Intellisense" desde el primer momento.

Interfaces

- Naturalmente, la interfaz, puede declarar métodos, y utilizar las técnicas habituales de sobrecarga:

```
interface Cosa2 {
    a: number;
    b: string;
      //funciones(métodos)
    hacerAlgo(s: string, n?: number): string;
    // Sobrecargas de métodos
    hacerAlgo(n: number): number;
}
```

Clases en TypeScript

- TypeScript nos permite trabajar con una de las novedades más importantes de ES6: las clases.
- Hasta la versión 5, JavaScript ha funcionado mediante herencia de prototipos.
- A partir TypeScript (y ES6, cuando esté soportado en los navegadores), podemos definir clases al estilo de lenguajes como Java/C#:

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
var greeter = new Greeter("world");
```

Clases en TypeScript: Implementación

- La implementación de clases permite incluir miembros de datos (estado), así como una parte funcional (métodos)
- Dentro del conjunto disponible de métodos posibles de usuario, hay uno cuyo nombre está asignado de antemano: ***constructor***.
- Será el método invocado con la sintaxis de llamada **new Clase()...**
- La declaración de un método un método constructor debe seguir todas las restricciones habituales de la POO al objeto de potenciar la coherencia de la clase.
- La palabra reservada ***this*** siempre se refiere a la clase activa.

Clases en TypeScript: Implementación

- Los métodos se definen como los de cualquier otra función:
  - Pueden recibir parámetros de cualquier clase
  - Se pueden definir parámetros que no son pasados si utilizamos las declaraciones ***opcionales.***

- Adicionalmente, muchos teóricos afirman que los principios S.O.L.I.D. deberían aplicarse igualmente en la implementación de una clase, y en especial, el primero (***Separation of Concerns***).

- Una clase podrá heredar de otra, sobrescribir métodos, implementar interfaces predefinidas, y servir de base a otras clases, al objeto de implementar una jerarquía.

Clases: Modificadores de Acceso

- Los ***getters/setters*** son una manera de interceptar cómo se accede a un miembro de un objeto.
- Esto permite un control más preciso sobre cómo se accede a un miembro de un objeto.
- Vamos a convertir una clase simple de usar ***'get'*** y ***'set'***. En primer lugar, vamos a empezar con un ejemplo base:

```
class EmpleadoBase {
    nombreCompleto: string;
}

var empleado = new EmpleadoBase();

empleado.nombreCompleto = "Joaquín Tillizo";

if (empleado.nombreCompleto) {
    alert(empleado.nombreCompleto);
}
```

Clases: Modificadores de Acceso

- Este acceso podría permitir modificaciones no deseadas. Para evitarlo creamos una versión de la clase con ***get/set*** que requiere una contraseña.

```
class EmpleadoGS {
   private _nombreCompleto: string;

   get nombreCompleto(): string {
      return this._nombreCompleto;
   }

   set nombreCompleto(nuevoNombre: string) {
      if (contraseña && contraseña == "contraseña") {
         this._nombreCompleto = nuevoNombre;
      }
      else { alert("Error: ¡Actualización no autorizada!");}
   }

}
```

BananaTube: Decisión en equipo

- Discute en equipo las ventajas de Typescript respecto a Javascript y ES6
- Sería una tecnologías por la que apostar?
- Tomad nota de las fortalezas y debilidades que hayas observado, así como, la dificultad de incorporarlo en el proyecto.

netmind

WeKnowIT

Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

GOBIERNO DE ESPAÑA
MINISTERIO DE ENERGÍA, TURISMO Y AGENDA DIGITAL
red.es
ESTRATEGIA DE EMPRENDIMIENTO Y EMPLEO JOVEN garantía juvenil
Agenda Digital para España
UNIÓN EUROPEA
Fondo Social Europeo
"El FSE invierte en tu futuro"