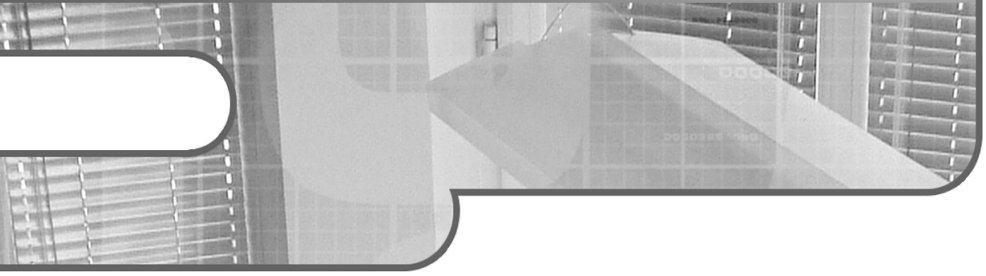


Capítulo 4
TypeScript

1. JavaScript

Desde hace algunos años, el lenguaje JavaScript se impone como el lenguaje indispensable en el desarrollo de aplicaciones web.

Inicialmente, JavaScript se creó como un lenguaje de scripting, utilizado para aportar un poco de dinamismo a los sitios web estáticos. Pero a lo largo del tiempo, con la ayuda de la popularización de frameworks como jQuery o AngularJS, la utilización del lenguaje JavaScript se ha hecho cada vez más importante, hasta permitir la creación de aplicaciones 100% front, con funcionalidades idénticas a lo que podemos hacer en una aplicación de cliente pesado (modo offline, notificaciones, etc.).

JavaScript es un lenguaje bastante diferente de lo que estamos acostumbrados a ver. En primer lugar, JavaScript es un lenguaje interpretado, el código escrito se ejecutad directamente sin fase de compilación inicial.

JavaScript también es un lenguaje dinámicamente tipado, es decir, que un elemento puede cambiar de tipo a lo largo de ejecución. De esta manera, a una variable de un determinado tipo, se le puede asignar un valor de otro tipo.

```
var miFuncion = function() {
    console.log("I'm a function");
}
miFuncion = 27;
```

Ejemplo del tipado dinámico JavaScript. La variable se inicializa como función y después recibe un valor entero.

Para terminar, JavaScript es un lenguaje por naturaleza mono-thread (mono-hilo), fuertemente asíncrono y basado en un mecanismo de eventos no bloqueantes. A pesar de su naturaleza mono-thread, JavaScript se ha diseñado como un lenguaje con muy buen rendimiento, que permite aprovechar la totalidad del rendimiento de los recursos que tiene asignados (al contrario que el modelo de lenguaje multi-thread (multi-hilo), que permite paralelizar un conjunto de operaciones).

Cuando se desarrolla una aplicación en JavaScript, su fuerte dinamicidad se convierte rápidamente en un hándicap. ¿Cómo se garantiza que el código escrito es sintácticamente válido?, ¿cómo realizar las fases de refactoring sin añadir regresiones?

```
var miFuncion = function() {
    console.log("I'm a function");
}
miFunction();
```

En el código anterior, hay un error de ortografía en la llamada al método `miFuncion`. El error solo será visible durante la ejecución del código.

```
var miFuncion = function(param) {
    console.log("I'm a function with a param " + param);
}
miFuncion();
```

En el código anterior, el método `miFuncion` se llama sin argumentos. Durante la ejecución, la consola visualizará el mensaje "`I'm a function with a param undefined`".

```
var miFuncion = function() {
    console.log("I'm a function");
}
miFuncion = 4;
miFuncion();
```

En el código anterior, se asigna un entero a la variable `miFuncion`, que era una función. La ejecución de este código provocará el siguiente error: "`miFuncion is not a function`".

Existe un conjunto de herramientas, como **jslint**, que permiten validar la calidad del código escrito pero no permiten sustituir a las verdaderas fases de compilación.

2. TypeScript

Una de las soluciones a estos problemas es la creación de lenguajes que se transcompilan en JavaScript, es decir, que la compilación genera archivos JavaScript. Este es principalmente el caso de TypeScript, actualizado por Microsoft y utilizado por Angular.

Existen otros lenguajes similares, como **Dart** o **CoffeeScript**, pero TypeScript es sin duda uno de los más completos en la actualidad.

Por lo tanto, TypeScript es un lenguaje compilado y tipado fuertemente, que genera JavaScript comprensible por todos los navegadores.

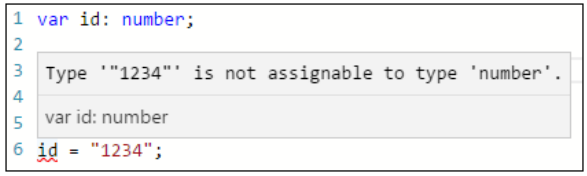
Observación

Hay otras soluciones, como la creación de nuevos lenguajes como Flash o Silverlight, que pueden ser interpretados por los navegadores gracias a la instalación de plug-ins. Estos lenguajes murieron cuando los navegadores decidieron no soportarlos. La ventaja de TypeScript es que genera código JavaScript totalmente comprensible por todos los navegadores de manera nativa, es decir, sin ayuda de plug-ins. Por lo tanto, esta solución es permanente en el tiempo.

El fuerte tipado de este lenguaje, va a permitir asociar un tipo a un elemento e impedir que este elemento cambie de tipo. Por lo tanto, este tipado fuerte permite una estabilidad superior del código, porque será posible predecir el tipo de un elemento y en consecuencia, sus diferentes valores posibles.

```
var id: number;
```

La fase de compilación de TypeScript permite validar sintácticamente el código. Si no es válido sintácticamente, es decir, si el desarrollador ha hecho uso de una propiedad en un objeto que no existe o que utiliza un tipo de manera no adecuada, la compilación del código TypeScript indicará este error. Las fases de refactoring, que eran extremadamente complicadas y peligrosas, ahora son mucho más sencillas. De manera general, el código será mucho más robusto y estable.



Hay numerosos editores, principalmente **Visual Studio**, **VSCode** o **SublimeText** por citar algunos, que incluyen el lenguaje TypeScript y ofrecen numerosas herramientas, como el autocompletado, los errores de compilación directamente en el editor, etc.

2.1 Sintaxis

La sintaxis de TypeScript es muy diferente de lo que estamos acostumbrados a ver en JavaScript. A continuación se muestra un resumen general de los principales elementos.

2.1.1 Variables

La declaración de una variable se hace de la siguiente manera:

```
var miVariableBooleana: boolean;
```

El tipo se define usando su tipo como sufijo en la definición de una variable, separado por ':'. Hay un conjunto de tipos básicos: `boolean`, `number`, `string`, `[]` para una tabla, etc.

Para conservar la posibilidad de beneficiarse del tipado dinámico de JavaScript, TypeScript introduce el tipo `any`.

```
var miVariableDinamica: any;
```

En esto caso, TypeScript no verifica el tipo de esta variable durante la compilación.

Si el tipo de una variable no se indica, TypeScript la considerará como de tipo `any`.

2.1.2 Funciones

La declaración de una función se hace de manera clásica.

```
function miFuncion(): void {
    ...
}
```

El tipo de retorno de la función, se define de manera idéntica al tipo de una variable. El tipo `void` indica que no hay ningún valor de retorno.

Si la función recibe argumentos de entrada, estos argumentos se tipan de manera idéntica a las variables:

```
function miFuncion(miArgumento: string, miOtroArgumento:
boolean): void {
    ...
}
```

2.1.3 Clases

La noción de clase apareció en JavaScript con EcmaScript 6. Esta noción existe desde hace varias versiones de TypeScript.

Observación

En la actualidad, la noción de clase introducida por EcmaScript 6 todavía no es compatible con todos los navegadores actuales, especialmente con Internet Explorer 11. Por lo tanto, todavía no es posible utilizarla si queremos apuntar a todos los navegadores actuales.

```
class Person {
    name: string;
    age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
```

```
    toString() {
        return `Hi I'm ${this.name} and I'm ${this.age} years
old!`;
    }
}
```

La palabra clave `class` permite indicar que vamos a crear una clase.

También es posible crear la herencia entre clases:

```
class Developer extends Person {
    constructor(name, age, language) {
        super(name, age);
        this.language = language;
    }

    toString() {
        return super.toString() + `:: I'm a Developer who likes
${this.language}`;
    }
}
```

La relación de herencia se declara con la palabra clave `extends`. A continuación es posible acceder a los elementos de la clase padre con el método `super`.

TypeScript también introduce la noción de visibilidad en las propiedades. Es posible declarar una propiedad `private`, que solo será accesible desde la clase actual `protected`, que solo será accesible desde la clase actual o las clases que heredan de la clase actual y para terminar una propiedad `public`, que será accesible desde el exterior de la clase. Por defecto, si no se ha especificado ninguna visibilidad, una propiedad será `public`.

```
class Modificar {
    public myPublicProperty: string;
    protected myProtectedProperty: string;
    private myPrivateProperty: string;
}
```