educación IT
CODE YOUR FUTURE

ReactJS Developer

Módulo 9

educación IT
CODE YOUR FUTURE

Introducción a *TypeScript*

TypeScript

TypeScript es un superconjunto de JavaScript. En otras palabras: ***TypeScript* es un lenguaje de programación construido sobre JavaScript con añadido de características típicas de lenguajes fuertemente tipados**.

TypeScript añade a JavaScript una fase extra en el proceso de construcción y una característica propia de los lenguajes compilados: que el código que yo escriba **no sea el que finalmente se ejecute**.

Usando JavaScript puro, sabemos que **el código que escribamos es el que finalmente se procesa**. Por otro lado, con TypeScript podemos aprovechar herramientas para mejorar la experiencia de desarrollo, sabiendo que luego no serán incluídas en el resultado final.

Comprobación de tipos primitivos

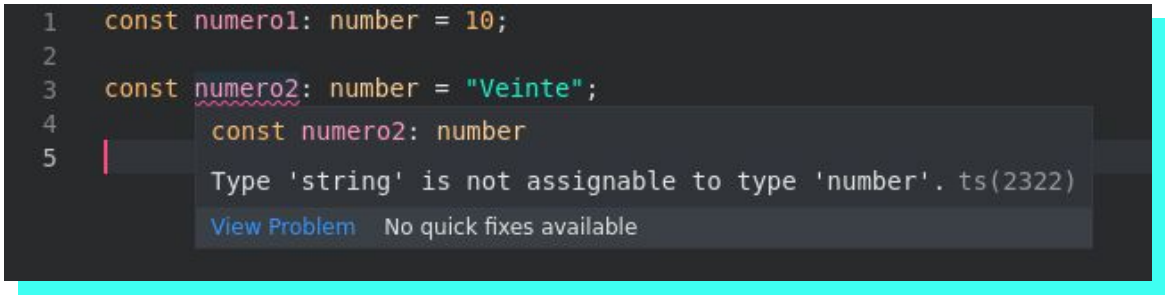
La comprobación de tipos estáticos (o *Type Check*) es la característica principal de *TypeScript*. Si bien ofrece más herramientas que el tipado dinámico, esta es la principal característica.

El *Type Check* **toma el tipo de dato de una variable**. Ese tipo de dato es **una palabra reservada precedida por** `:` (dos puntos) **luego del nombre de la variable**.

En las próximas slides veremos cómo *TypeScript* realiza el *Type Check* de los tipos primitivos.

Comprobación de tipos primitivos *Number*

El tipo `Number` **verifica que el contenido de esa variable sea cualquier número** (entero o flotante):



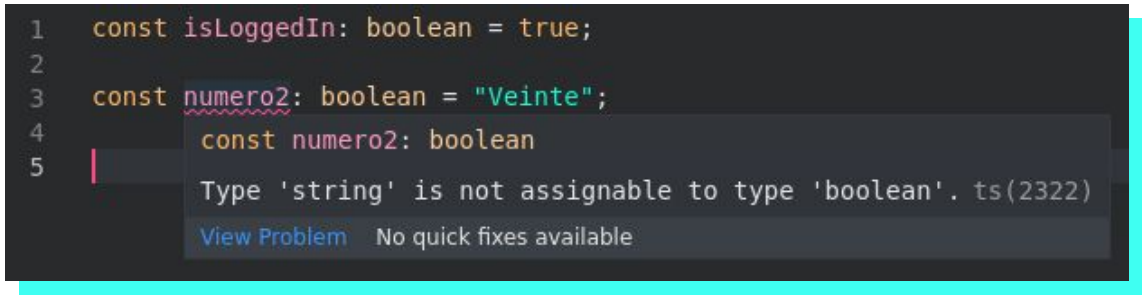
Comprobación de tipos primitivos *String*

El tipo `string` verifica que el contenido de esa variable sea **un texto alfanumérico (encerrado entre comillas)**:

```
1  const texto: string = "Hola mundo";
2
3  const numero: string = 20;
4          const numero: string
5
6          Type 'number' is not assignable to type 'string'. ts(2322)
           View Problem    No quick fixes available
```

Comprobación de tipos primitivos *Boolean*

El tipo `boolean` verifica que el contenido de esa variable sea **`true`, `false` o cualquier expresión equivalente**:



Comprobación de cabeceras de función

Siguiendo con la misma lógica de verificación de tipos, podemos indicar al *Type Check* que lance error **si una función no cumple con una firma determinada**.

Esta verificación nos sirve para asegurarnos que una función **cumpla con estas características**:

```
1  const sumar: (n1: number, n2: number) => number = function (n1, n2) {
2      return n1 + n2;
3  };
```

Recordemos que la firma de la función es **el juego de parámetros que debe tener y el tipo de dato que debe retornar**. Se escribe de la siguiente forma:

```
(param: tipo, param2: tipo) => tipo_retorno
```

Existen funciones que **no retornan nada** y para eso se puede usar el tipo especial **`void`**:

```
(param: tipo, param2: tipo) => void
```

También podemos especificar funciones que **no tengan parámetros**:

```
() => tipo_retorno
```

Comprobación de propiedades de objetos (interfaces)

Los objetos se validan por el *Type Check* a través de una estructura propia denominada **interfaz**. La interfaz **actúa a modo de "contrato" entre el objeto y aquellos que lo utilicen**.

¿Qué significa esto? **Que cuando un objeto use una interfaz** (es de "su tipo") **estará obligado a respetar la estructura de esa interfaz**, es decir, a tener todas las propiedades obligatorias y a respetar los nombres y tipos de dato definidos en ese contrato.

Para el momento de desarrollo, esto tiene la enorme ventaja de que nos permite **hacer predecible la estructura de un objeto y darla a conocer por toda la aplicación**.


interface Post {
title: string;
body: string;
}
const miPost: Post = {
body: "",
title: ""
}
miPost.
body (property) Post.body: string
title

Tipos genéricos

Los tipos genéricos permiten al *Type Check* ser **un poco más flexible**. Una declaración de tipo genérico actúa como "*placeholder*" de tipos de datos ¿Qué significa esto? Que el *genérico* **tiene un tipo de dato que la estructura desconoce y que se definirá según su uso**.

Se usa mucho en funciones **que deben ser lo suficientemente flexibles como para cambiar de tipo de dato**.

Un ejemplo interesante sería la solicitud de datos a *API*, en la que podemos hacer el mismo proceso para recibir **Usuarios** o **Posts**.

Para usar un tipo genérico, debemos encerrar su definición entre **<** y **>**. El contenido que esté ahí dentro será un **identificador**. Cuando el usuario llame a esta función, **deberá llamarla de igual forma y pasar un tipo de dato**. Ese tipo de dato será **el usado por el *placeholder***.

Si llamo a la siguiente función de esta forma, obtendré una promesa de un *array* de números:

```
loadResource<number[]>('/stats')
```

```
async function loadResource<ResultType>(path: string): Promise<ResultType> {
    const response = await fetch(`https://misitio.com/api${path}`);
    return response.json();
}
```

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!