SOFTRADIS

TypeScript

JavaScript with syntax for types

Conceptos TypeScript

¿Qué es TypeScript?

TypeScript es un lenguaje de programación libre, de código abierto desarrollado y mantenido por Microsoft. Esencialmente añade tipos estáticos y objetos basados en clases. Su principal uso es el desarrollo de aplicaciones JavaScript que se ejecutarán en el lado del cliente o del servidor, y/o extensiones para programas.

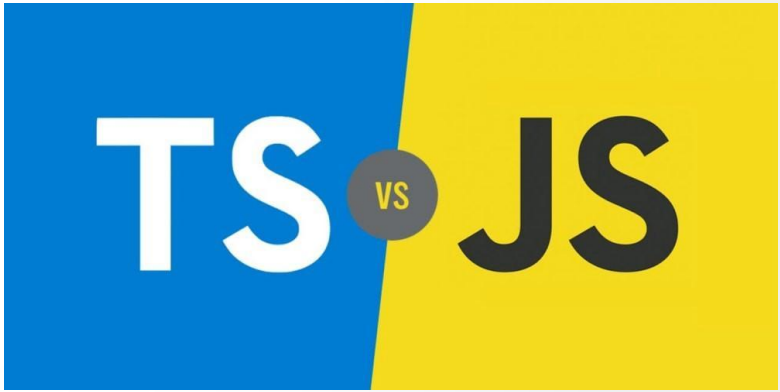
¿Por qué usar TypeScript?

1. Es uno de los lenguajes de programación más queridos por la comunidad.
2. Alta adopción.
3. Podemos prevenir muchos errores de JS.
4. Mejor experiencia de desarrollo.
5. Menor cantidad de bugs.

TypeScript vs JavaScript

En JavaScript te das cuenta de los errores del código una vez que lo estés ejecutando. En TypeScript con el análisis estático de código nos damos cuenta de los errores en el editor o a la hora de transpilar, por lo que evitamos que los errores lleguen a la ejecución.

Un programador de TS es distinto a uno de JS. El programador de TypeScript añade una capa de análisis de código estático a Javascript.



¿Cómo se pueden encontrar errores?

En el libro **Software Engineering at Google** lo dividen en distintas etapas:

1.- Static analysis

2.- Unit Tests

3.- Integration tests

4.- Code review

```
(()=> {
  const myCart = [];
  const products = [];
  const limit = 2;

  async function getProducts() {
    const rta = await
fetch('<http://api.escuelajs.co/api/v1/products>', {
      mehtod: 'GET'
    });
    const data = await rta.parseJson();
    products.concat(data);
  }
  function getTotal() {
    const total = 0;
    for (const i = 0; i < products.length(); i++) {
      total += products[i].prize;
    }
    return total;
  }
  function addProduct(index) {
    if (getTotal <= limit) {
      myCart.push(products[index]);
    }
  }
  await getProducts();
  addProducto(1);
  addProducto(2);
  const total = getTotal();
  console.log(total);
  const person = {
    name: 'Nicolas',
    lastName: 'Molina'
  }
  const rta = person +  limit;
  console.log(rta);
});
```

```
//@ts-check

(async ()=> {
  const myCart = [];
  const products = [];
  const limit = 2;

  async function getProducts() {
    const rta = await
fetch('<http://api.escuelajs.co/api/v1/products>', {
      method: 'GET'
    });
    const data = await rta.json();
    products.concat(data);
  }
  function getTotal() {
    let total = 0;
    for (let i = 0; i < products.length; i++) {
      total += products[i].prize;
    }
    return total;
  }
  function addProduct(index) {
    if (getTotal() <= limit) {
      myCart.push(products[index]);
    }
  }
  await getProducts();
  addProduct(1);
  addProduct(2);
  const total = getTotal();
  console.log(total);
  const person = {
    name: 'Nicolas',
    lastName: 'Molina'
  }
  const rta = `${person} ${limit}`
  console.log(rta);
});
```

Atrapados Errores

Un archivo de **Javascript,** puede pasar desapercibido los errores que tiene, pero cuando lo analizamos con **Typescript** salen a la luz y permite que los detectemos al inicio.

¿Qué es el tipado en TypeScript?

Si bien Javascript tiene tipos, es un lenguaje débilmente tipado, por lo que no se realiza una comprobación de los tipos. Al hacerlo tipado con TypeScript, evitamos errores ya que el analizador comprueba que no cambiemos los tipos a lo largo del tiempo.

A la hora de **declarar variables** en Javascript lo hacemos de la siguiente forma:

```
const productPrice = 12;
```

Mientras que en **TypeScript** tenemos que especificar los tipos:

```
const productPrice:number = 12;
```

La parte del tipado, la llamamos Type Annotation.

Tipos Inferidos

El motor de **TypeScript** nos permite tanto darle el tipo de datos de forma explícita o los puede inferir por sí mismo sin que lo tengamos que hacer de forma explícita.

```
let myProductName = 'Product'; //Nos dice que el tipo es string
let myProductPrice = 123; //Nos dice que el tipo es number
```

```
.toLowerCase
```

```
(() => {
    let myProductName = 'Product';
    let myProductPrice = 123;
})();
```

Al no tener el entorno de **TypeScript** configurado ni delimitado el alcance de las funciones, al hacer una declaración en un archivo distinto con el mismo nombre, nos va a arrojar error. Para delimitarlo, podemos hacer una función anónima autoejecutada y encerrar nuestro código ahí y que su scope esté limitado a esa función.

Tipos de Datos

Numbers

Para utilizar el primitivo number de forma explícita, lo podemos tipar con **:number**

```
let productPrice: number = 100
productPrice = productPrice + 1
productPrice = productPrice + '1'
// Esto nos lo alerta, ya que JS
lo transformaria a string.
```

Debemos asignarle a la variable el tipo ya que TS no lo puede inferir.

```
let productInStock: number;
console.log(productInStock)
// TS nos alerta que no
inicializamos nuestra variable.
```

parseInt nos convierte el string a number:

```
let discount = parseInt('123')
// TS infiere que discount es
number.
```

TS nos permite utilizar otro tipo de números::

```
let hex = 0xFFF //Utilizamos 0x
para denominar a los
hexadecimales.
let bin = 0b1010 //Utilizamos 0b
para demnominar a los binarios.

console.log(hex) //4095
console.log(bin) //10
```

Recuerda: **number** y no **Number**

Tipos de Datos

Booleans

Nos permite utilizar los booleanos ***true*** y ***false***

```
let isEnabled = true;
//isEnabled = "hello" //Da error, ya que TS infirio que el dato es de tipo boolean

let isNew : boolean = false;
console.log(isNew) //false
isNew = true;
console.log(isNew) //true

let isNew: boolean;
const random = Math.random();
//isNew = random >= 0.5 ? "true" : "false" Esto da error ya que se le asignaria un
string, tampoco podemos asignar 1 y 0 para ture y false ya que son tipo number
isNew = random >= 0.5 ? true : false
```

Recuerda: **boolean** y no **Boolean**

Tipos de Datos

Strings

Al igual que los anteriores, lo podemos hacer de forma explícita con ***:string*** o de forma implícita.

```
let productTitle = 'My amazing
product';
// productTitle = null;
// productTitle = () => {}
// productTitle = 123;
productTitle = 'Newest amazing
product'

let productDescription: string =
'Awesome description'
productDescription = "I'm a
description (?"
```

También podemos utilizar los template literals para los strings:

```
let productTitle:string =
'Newest amazing product';
let productDescription: string =
'Awesome description';
let productPrice: number = 123;

let summary:string = `
title: ${productTitle}
description:
${productDescription}
price: ${productPrice}`
```

Tipos de Datos

Arrays

En los arrays podemos definirlos de forma **implícita** como **explícita:**

```
let prices =
[123,312,643,124,12,75] //Aca
infiere que el tipo de dato
dentro del array es number
//prices.push('asd')
//prices.push(true)
//prices.push({12})
prices.push(111)

let myArr = [123,523,15,'Hola',
true] // Aca infiere que los
tipos son boolean, number y
string
```

Para hacerlo de **forma explícita:**

```
let mixed: (number | string | boolean)[] = ['Hola', true];
mixed.push(123)
mixed.push('mundo')
mixed.push(true)
// mixed.push({})
// mixed.push([])
```

Tipos de Datos

Any

* Este tipo de dato desactiva el tipado de TS y nos permite re-asignar el tipo de dato de una variable.

* No está recomendado utilizar ***any,*** debido a que perdemos la posibilidad de atrapar cualquier error.

```
let myDinamicVar : any;
myDynamicVar = 100;
myDynamicVar = "Hola";
myDynamicVar = null;
myDynamicVar = {};
```

* Podemos convertir el tipo de ***any*** a cualquier otro tipo de dato haciendo un **cast** con el ***as*** operator o utilizando genéricos ***<type>***.

```
let myDynamicVar = 'Hola';
const rta = (myDynamicVar as string);
console.log(rta);
```

Tipos de Datos

Union Types

Nos permite mayor flexibilidad en el código, ya que posibilita asignar varios tipos a una variable.

```
let userID: string | number;
userID = 1212;
userID = 'FC45';

function greeting(myText: string | number) {
  if (typeof myText === 'string') {
    console.log(`Hola ${myText.toLowerCase()}`);
  } else if (typeof myText === 'number') {
    console.log(`Sos el usuario Nº${myText}`);
  }
}
/**Cuando pasa los if, nos habilita el
autocompletado por tipos**/
userID = 'FC45';
greeting(userID);
```

Tipos de Datos

Alias y Tipos Literales

Los **alias** nos permiten agrupar varios tipos dentro de un prototipo.

```
type User = string | number;

let userID: User;
let userName: User;

userID = 12345;
userName = 'Krissel'
```

Los **literal types** nos permiten limitar dentro de una variable los valores.

```
type ShirtSizes = 'S' | 'M' |
'L' | 'XL';
let shirtSize: ShirtSizes;
shirtSize = 'M'
```

```
let myNull = null; //Infiere any
let myUndefined = undefined; //Infiere any

let myName: string | null; //Hay veces que
necesitamos que una variable

function hi(name: string | null) {
  let hello: string = 'Hello ';
  if (name) {
    hello += name;
  } else {
    hello += 'nobody';
  }
  console.log(hello);
}
myName = null;
hi(myName); // "Hello nobody"
myName = 'Franco';
hi(myName); // "Hello Franco"
```

Tipos de Datos

Null y Undefined

Para poder usar Null o Undefined en una variable, lo tenemos que especificar con los Union Types, ya que tanto *null* como *undefined* son su propio tipo de dato.

Tipos de Datos

Null y Undefined

También esto lo podemos hacer de la siguiente manera con **optional chaining:**

```
function hi(name: string | null) {
  let hello: string = 'Hello ';
  hello += name?.toLowerCase() ||
'nobody'
  console.log(hello);
}
```

La usamos cuando no tenemos la certeza de si tenemos un valor o no.

También lo podemos hacer con el **Nullish Coalesing** de la siguiente forma:

```
function hi(name: string | null) {
  let hello: string = 'Hello ';
    hello += name ?? 'nobody'
  console.log(hello)
}
```

Funciones

En **TS** tenemos que declarar el tipo de dato de cada parámetro en la función. Además de que vamos a tener la referencia de si nos faltan o sobran parámetros.

```ts
type ShirtSizes = 'S' | 'M' | 'L' | 'XL';
function createProductToJSON(
title: string,
createdAt: Date,
stock: number,
size: ShirtSizes){
  return {
    title,
    createdAt,
    stock,
    size
  }
}

const product1 =
createProductToJSON('Prod 1',new Date(),
12,'XL')
console.log(product1)
console.log(product1.stock)
```

También podemos usar **arrow functions** y **parámetros opcionales** que de no ser declarados van a ser undefined.

```ts
type ShirtSizes = 'S' | 'M' | 'L' | 'XL';
const createProductToJSON = (
  title: string,
  createdAt: Date,
  stock: number,
  size?: ShirtSizes
) => {
  return {
    title,
    createdAt,
    stock,
    size,
  };
};
```

Retorno de Funciones

Con TypeScript podemos especificar el tipo de dato de lo que nos va a retornar agregandolo luego de los parámetros o lo puede inferir por sí solo.

```
const calcTotal = (prices: number[]):number => {
  let total:number = 0;
  total = prices.reduce((acc,num) => acc +
num,0)
    return total
}
const rta = calcTotal([1,2,3,4])
```

Void

Nos permite especificar el tipo en funciones que no retornan nada.

```
const printTotal = (price:
number):void => {
    console.log(`El total es:
$${price}`)
}
```

Objetos de Funciones

Al pasarle como parámetro objetos a typescript este nos permite especificar tanto las keys como los tipos de cada value.

```
const login = (data: {email: string, password: string}):void => {
  const {email, password} = data
  console.log(email,password)
}
login({email: "fran@co.me", password: "N0M3H4CK335"}
```

product.model.ts

```ts
export type ShirtSizes = 'S' | 'M' | 'L' | 'XL';
export type Product = {
  title: string;
  createdAt: Date;
  stock: number;
  size?: ShirtSizes;
};
```

product.service.ts

```ts
import { Product } from '.product.model.ts'

export const products: Product[] = [];
export const addProduct = (data: Product): void => {
  products.push(data);
};
export const calcStock = ():number => {
  let total = 0;
  products.forEach(product => total += product.stock)
  return total;
}
```

Módulos y Librerías

Módulos: Import y Export

Como buena práctica, a los archivos para modelado de datos debemos llamarlos de la siguiente forma: ***nombre.model.ts*** y a los archivos que van a tratar dichos datos de la siguiente manera: ***nombre.service.ts***

product.ts

```
mport { addProduct, products, calcStock}
from './product.service.ts'


addProduct({
  title: 'Camisa',
  createdAt: new Date(),
  stock: 10,
  size: 'M',
});
addProduct({
  title: 'Gorra',
  createdAt: new Date(),
  stock: 15
});

console.log(products)
const total = calcStock()
console.log(total)
```

Módulos y Librerías

Módulos: Import y Export

Conclusiones

TypeScript nos ayuda a encontrar los errores que aparecen, ya que va analizando el código constantemente a medida que lo vamos escribiendo y va alertando en caso de requerir, además de que ayuda a la legibilidad por saber que tipo de datos reciben y retornan las funciones, solo leyendo el tipado.