Friday, March 31, 2017

TypeScript

What is it ?

TypeScript brings more features to ES6, but like we talked before we need to transpile it to ES5 so it can run on most browsers.

The most important feature that TypeScript brings is the addition of type information, it helps the developer in creating a more secure and bulletproof code giving the possibility to add types to variables just like in other languages like JAVA for example, and this way controlling parameters and other aspects of coding.

How to use?, you just need to add “: “ followed by the type you want. Example:

```
function howMuch(a: number ) {
        return `the number is ${a}`;
}
```

if we enter something that is not a number, for example : howMuch(‘a number’) its going to fail on compiling before Js is even produced.

Now, different that we currently do by just creating variables on the go, TypeScript is more demanding and we need to declare the instance properties.

// Before:

```
class TestClass {
	constructor( testVariable) {
		this.testVariable = testVariable;
	}
	modifyValue(newValue) {
		this.testVariable = newValue;
	}
}
const testClass = new TestClass(2);
testClass.modifyValue(“number 2”);
```

This works !

```
// After:

class TestClass {
	testVariable: number;
	constructor( testVariable: number) {
		this.testVariable = testVariable;
	}
	modifyValue(newValue: number) {
		this.testVariable = newValue;
	}
}
const testClass = new TestClass(2);
testClass.modifyValue(“number 2”);
```

This doesn't works !

TypeScipt Features

Types

Besides “boolean, number, string, [], {}, and undefined that javascript already provides, TypeScript adds “enum (enumerations like {Red, Blue, Green}), any (use any type), and void (nothing).

Classes

You can use a class as a type, this way :

```
class Foo { foo: number; }

class Bar { bar: string; }

class Paz {

        constructor(foo: Foo, bar: Bar) {}

}

let paz = new Paz( new Foo(), new Bar());
```

You can use “?” key to let know that is an optional parameter, example :

```
class ExampleClass {

        variable1: string;

        variable2?: string;

}
```

Interfaces

Its a TypeScript artifact, not part of ECMAScript, is an abstract type which doesn't have any code inside, it defined the shape of an API. During transpilation, an interface will not generate any code but is used by TypeScript to type checking during development.

Example:

```
interface Callback {
        (error: Error, data: any): void;
}

function callServer( callback: Callback) {
        callback(null, ‘hi’);
}
callServer( (error, data) => console.log(data)); // ‘hi’
callServer(‘hi’) // gives an error
```

It also helps to define a variable with more than one type, example:

```
interface PrintOutput {
	(message: string): void;
	(message: string[]): void;
}
let printOut: PrintOutput = (message) => {
	if(Array.isArray(message)) {
		console.log(message.join(‘,  ‘));
	}else {
		console.log(message);
	}
}

printOut(‘hello’); // ‘hello’
printOut([‘hello’, ‘bye’]) // ‘hello, bye’
```

Example if interface describing an object literal:

```
interface Person {
	name: String;
}

let person: Person = {
	name: ‘Alan’
}
```

Shapes

They work like TypeScript interfaces, and are in fact how TypeScript compares custom types like “classes” and “interfaces”.

For example taking the last example:

```
interface Person {
    howOld: String;
}
let person: Person = {
    howOld: ‘not too much i hope’
}
```

we can do something like this:

```
class notAPerson {
    howOld: string;
    constructor() {
        this.howOld = “a lot”;
    }
}

person = new notAPerson();
```

And this is valid !!, because they have the same “Shape”, if two objects have the same attributes, with the same typings, those two objects are considered to be the same type.

Type Inference

We don't need to describe a type ALWAYS, TypeScript can infer the type of an object.

Example:

```
let aCollectionOfNumbers = [ 1, 2 , 3 , 4 , 5];
aCollectionOfNumbers = [`this will generate an error`];
```

TypeScript in this case infers that aCollectionOfNumbers is an array so when we try to assign a script to it an error is generated.

Type keyword

We can use “type” to define an alias to one, example:

```
type str = string;
let variable1: str = ‘this is a string and works!’;
let variable2: str = 1 // this doesn't
```

Union Types

This feature helps us to define more than one possible type without the using of interfaces, example:

```
function unionExample (exampleVariable: number | string) {
        return `this variable equals ${exampleVariable} and is valid`;
}
```

unionExample(30); // “this variable equals 30 and is valid”;

unionExample(“a really nice variable”); // “this variable equals a really nice variable and is valid”;

And to improve things lets use the “type” keyword:

```
type exampleType = number | string;
function unionExample (exampleVariable: exampleType) {
        return `this variable equals ${exampleVariable} and is valid`;
}
```

Intersection Types

These are combinations of two or more types, which is really useful for objects and params that need to be implemented at more than one interface.

```
interface Sit {
        sit(howLong: number): number;
}
interface Stand {
        stand(howHig: number): number;
}
// iCan is an intersection between Sit and Stand, having both properties.
type iCan = Sit & Stand;
function justDoIt(shiaLaBeouf: iCan ) {
        shiaLaBeouf.sit(10);
        shiaLaBeouf.stand(20);
        shiaLaBeouf.doIt(); // ‘doIt’ des not exist on type ‘iCan’.
}
```

Function type definitions

They are more specific than “Function” type that we normally use, for example:

```
type canHappenToAnyone = Error | null;
type Callback = (err: canHappenToAnyone, response: Object) => void;

function sendRequest (cb: Callback): void {
    if(cb) {
        cb(null, {});
    }
}
```

Decorators

They are a special kind of declaration that can be attached to a class, method, accessor, property or parameter. You can use them by the “@“ keyword, next to the expression with represents a certain function, and with that we can change the functionality of it.

You can learn more here.

Example:

```
@myLovelyFunction;
```

Where myLovelyFunction is:

```
function myLovelyFunction( target) {
    // do something with “target”
}
```