Basic types

Learn the basics of TypeScript and how to define simple types

WE'LL COVER THE FOLLOWING

An Intro To `TypeScript`

Now that you have a clear idea of what JavaScript looks like in 2019, I think it's time to introduce you `TypeScript`.

Albeit not necessary as a skill for a JavaScript developer, I believe it to be extremely useful, especially when working in team on bigger projects.

As you already know, JavaScript is not a **strongly typed** language meaning

that you don't have to define the type of your variables upon declaration.

This means that they can be more flexible and accept different values. At the same time it can make the code more confusing and prone to having bugs.

Look at this example:

```
function getUserByID(userID){
  // perform an api call to your server to retrieve a user by its id
}
```

What is `userID` ? Is it an `integer` or a `string`? We can assume it's an integer, but maybe it's an alphanumeric string (e.g.: 'A123').

Unless we wrote that piece of code, we have no way of knowing what the type of the argument is.

That's where `TypeScript` comes in handy. This is how the same code would look:

```
function getUserByID(userID:number){
  // perform an api call to your server to retrieve a user by its id
}
```

Perfect! Now we know for sure that if we pass a string to the function, that will cause an error.

A simple addition made the code easier to use.

What is `TypeScript`?

Created just a few years ago by Microsoft, TypeScript is a typed superset of `JavaScript` that compiles to plain `JavaScript`.

Being a superset means that you can write plain `JavaScript` in a `TypeScript` file and no errors would occur.

Browsers don't understand `TypeScript`, which means it has to be transpiled to plain `JavaScript`.

Let's look at how to set up our environment to write `TypeScript`.

How to use `TypeScript`

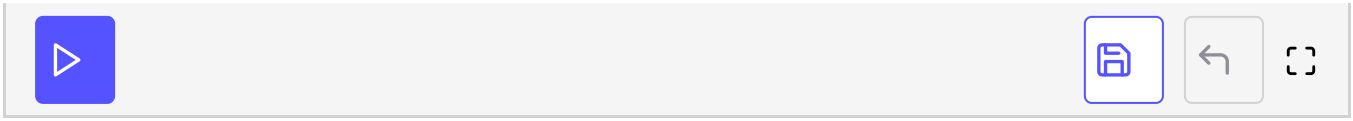
Getting started with `TypeScript` takes literally 5 minutes.

The first thing to do is to install it. Run this command in your terminal:

```
npm install -g typescript
```

Next, you can open your code editor and create a file called `greeter.ts` (NOT `.js`).

```
const greeter = (name:string) => {
    console.log(`hello ${name}`)
}
greeter('Alberto');
// hello Alberto
```

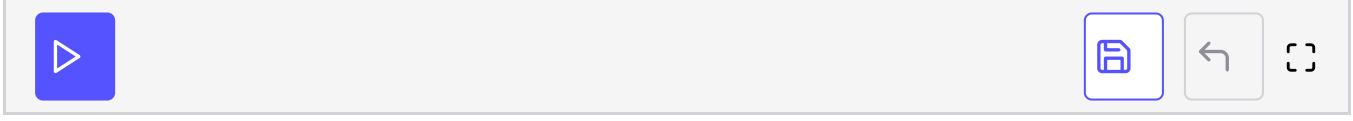


Now what we have to do to generate the `JavaScript` file is run this command in the terminal in the same directory as our file:

```
tsc greeter.ts
```

Now you should have a `greeter.js` file with this code inside:

```
var greeter = function (name) {
    console.log("hello " + name);
};
greeter('Alberto');
// hello Alberto
```



Where did `name:string` go? As we know, JavaScript is not strongly typed. Therefore, the type declaration get removed when the code is transpiled.

Typing your code will help *you* to debug it and to cause less errors, but it won't create a different JavaScript output than what you would have gotten if

you didn't use types.

TypeScript basic types

In this section we'll go over the basic types supported by TypeScript. You should already know the meaning of most of them from the `Introduction to JavaScript` section.

The basic types supported by `TypeScript` are:

- `boolean`
- `number`
- `string`
- `Array`
- `object`
- `Tuple`
- `enum`
- `any`
- `void`
- `null` and `undefined`
- `never`

`boolean`

Defines a value that can be true or false:

```
const active: boolean = true;
```

`number`

The type `number` supports hexadecimal, decimal, binary and octal literals:

```
const decimal: number = 9;
```

```
const hex: number = 0xf00d;
const binary: number = 0b1010;
const octal: number = 0o744;
```

`string`

The type `string` is used to store textual data:

```
const message: string = 'Welcome";
```

`Array`

There are two ways of defining `Array` types:

```
// first way -> type[]
const firstArray: number[] = [1,2,3]

// second way -> Array<type>
const secondArray: Array<number> = [4,5,6]
```

In simple cases like this one, there's no difference between them. But if we are working with something more complex than a type of number, then we won't be able to use the first notation.

Look at this example:

```
// our function accepts an array of  objects with a label and a value as its properties
function example(arg: Array<{label:string,value:string}> ){
  // do something
}
```

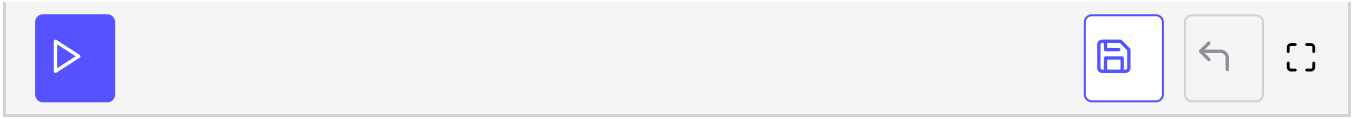
`Array<{label:string,value:string}>` simply means that the argument is an `Array` of objects with properties of label and value, both of type string.

`object`

`object` represents a value that is not one of the primitives;

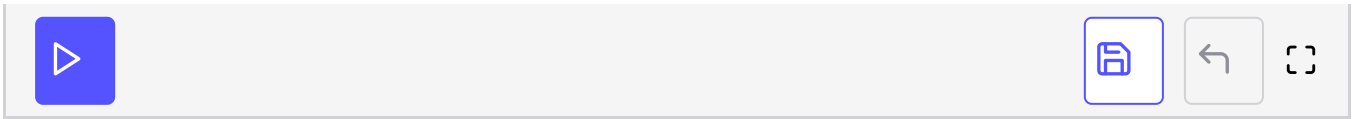
Let's say our function takes an argument of type `object`:

```
function greetUser(user: object){
  // property name does not exist on type object
  console.log(`hello ${user.name}`)
}
greetUser({name: 'Alberto', age:27});
// hello Alberto
```



The `Typescript` compiler will complain that the property `name` does not exist on the type `object`. Let's define better the properties of that object.

```
function greetUser(user: {name:string,age:number}){
  console.log(`hello ${user.name}`)
}

greetUser({name: 'Alberto', age:27});
// hello Alberto
```

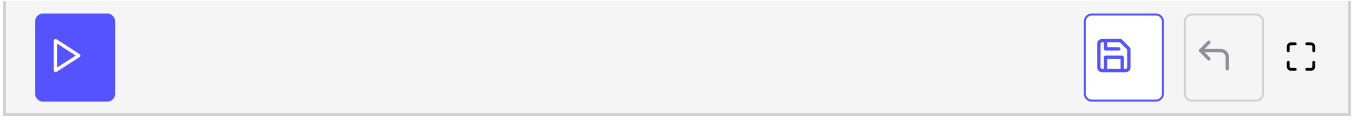


Now we specified all the properties of the user `object`, making it easier for us and for anybody who looks at the code to know what they can and what they cannot do with that `object`.

`Tuple`

A `Tuple` allows you to define the type of the known elements of an array.

```
let myTuple: [string,number,string];
myTuple = ['hi',5,'hello']

console.log(myTuple);
// [ 'hi', 5, 'hello' ]
```

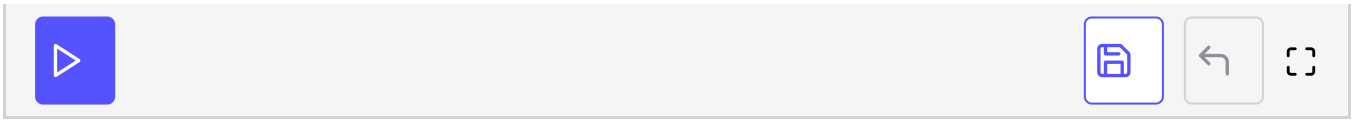


`TypeScript` will know the type of the elements of the indexes that we define in the tuple, but it won't be able to know the type of additional elements added to the array.

`enum`

An `enum` is a way of giving names to a set of numeric values:

```
enum Status {deleted, pending, active}

const blogPostStatus: Status = Status.active;

console.log(blogPostStatus);
// 2
```

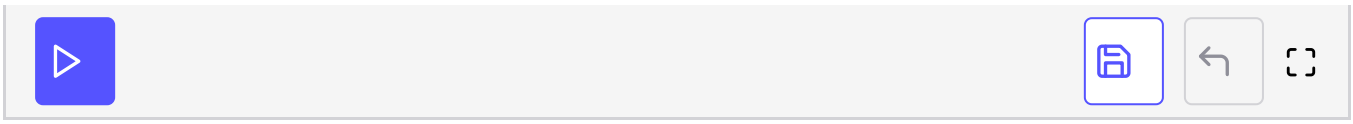


The values inside of an enum start from 0 so in our example before `active` corresponds to 2, `pending` to 1 and `deleted` to 0.

It is much more meaningful to say that the status of a blog post is `active` rather than 2.

You could override the starting point of an enum by specifying it like this:

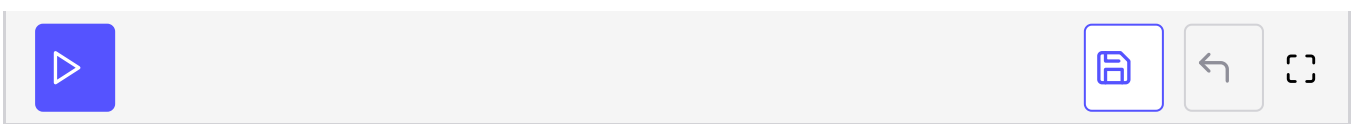
```
enum Status {deleted = -1, pending, active}

const blogPostStatus: Status = Status.active;
console.log(blogPostStatus);
// 1
```



Now `deleted` corresponds to -1, `pending` to 0 and `active` to 1, much better than before.

We can also access the values of an enum based on their value:

```
enum Status {deleted = -1, pending, active}
console.log(Status[0]);
// pending
```



`any`

As the name implies, `any` means that the value of a certain variable can be anything. We may use it when dealing with 3rd party libraries that don't support `TypeScript` or when upgrading our existing code from plain `JavaScript`.

`any` allows us to access properties and methods that may not exist.

We can also use `any` when we only know part of our types:

```
let firstUser: Object<any> = {
  name: 'Alberto',
  age: 27
}

let secondUser: Object<any> = {
  name: 'Caroline'
}
```

We expect both variable to be of type `object`. But we are not sure of their properties, therefore we use `any`.

`void`

`void`, as the name implies, defines the absence of type. It's often used in scenarios like this:

```
function storeValueInDatabase(objectToStore): void {
  // store your value in the database
}
```

This function takes an `object` and stores it in our database but does not return anything. That's why we gave it a return value of `void`.

When declaring variables of type `void` you will only be able to assign values of `null` and `undefined` to them.

`null` and `undefined`

Similarly to `void`, it's not very useful to create variables of type `null` or `undefined` because we would only be able to assign them `null` and `undefined` as values.

When talking about *union types* you will see the use of these two types.

`never`

`never` is a value that never occurs. For example we can use it for a function that never returns or that always throws an error.

```
function throwError(error:string): never{
  throw new Error(error)
}
```

This function only throws an error, it will *never* return any value.

In the next lesson, we'll cover interfaces and classes.