



Juan Guillermo Zafra Fernández

Jorge Quintana García



# Content

- ◇ 1- What is TypeScript?
- ◇ 2- How to get started?
- ◇ 3- Basic use
- ◇ 4- Types
- ◇ 5- Classes
- ◇ 6- Interfaces
- ◇ 7- Extra things
- ◇ 8- Conclusions





# Before we get started

We make usage of some lingo:

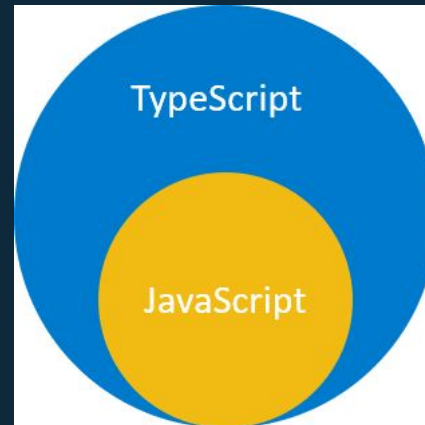
- Extend === Inherit
- TS === TypeScript
- JS === JavaScript





# 1- What is TypeScript

- ◆ TypeScript is a super set of JavaScript.
- ◆ Adds additional syntaxes for supporting Types.
- ◆ Compiles from TS to JS





## 2- How to get started?

◇ You will need only a few things:

- Node.js.
- TypeScript Compiler.
- Text Editor.





# Installing TypeScript

You will have to enter these commands in the terminal:


1. `$npm install -g typescript`
2. `$tsc --v` (to check the version installed)
3. `$npm install -g ts-node`

In Windows is possible to get an error implying it does not recognise 'tsc'.

To solve the error, add C:\Users\<user>\AppData\Roaming\npm to the PATH variable.

Is possible to get an error using the keyword console (for example in console.log).

To solve the error, run the following command: `$npm install @types/node --save-dev`





## 3- Basic usage

After installation you are ready to code:

1. Write your TS file
2. Run `$tsc app.ts` where `app.ts` is your file
3. If everything is correct then it should appear a `.js` file with the same name
4. Run `$node app.js`

You can run it directly running `$ts-node app.ts`





## 4- Types

Types are half the story about TS.  
It lets the programmer specify what  
type a variable is, and avoids  
mistakes thanks to a handy  
compiler.

The friend for hard-typed  
languages fans.







# *Types overview*

```
1  let a: number = 1;
2  let b: string = "A";
3  let c: boolean = true;
4  let d: number[] = [1, 2];           // array
5  let e: [number, string] = [1, "A"]; // tuple
6  let f: (number | string) = 1; // or "A" // union
7  let g: any = "A"; // or 1, [1, 2], ... // any
8  let h: void = undefined; // or null // void
9  let i: never; // nothing can be assigned // never
10
11 let j: {x: number, y: string} = {x:1, y:"A"}; // object
12
13 function hello(name: string): void {           // function
14 |   console.log(`Hello, ${name}!`);
15 }
16 let k: (x: string) => void = hello;           // function object
```





# *Types of types*

- ◇ Primitive
  - number
  - string
  - boolean
- ◇ Objects
  - Everything else





# *Type inference*

TS can inference the type of most things for you.

Type inference	Type annotation
TS guesses the type	You specify the type

For the most part, let TS guess. Only specify:

- ◇ If you want better readability.
- ◇ If TS can't infer the type itself.
- ◇ Declare a variable for later use.





# *Objects and Tuples*

Objects work as expected in TS as a type, but the compiler also checks when you access a property if the object has it.

Tuples are arrays where:

- ◇ The length can't be altered.
- ◇ The type of the elements within is known.
- ◇ The order matters





## *Enum*

Very much alike Python's enumerations. It's a set of values that are related in some way.

They go well with constants.

## *Any*

Any represents a type that can be any type.  
Simple.





## *Void & Never*

**Void** denotes the *absence* of any type.

It's often used to mark that a function returns nothing.

**Never** is used to mark that you can't assign to anything.

It's used to mark functions that throw errors, or loop indefinitely.





# *Type joining & Aliases*

You can join two types into one using the | operator.

You can make an alias of an existing type (renaming it).



```
let result: number | string;  
result = 10; // OK  
result = 'Works'; // OK  
result = false; // One way ticket to compiling error
```





# *String literals*

Instead of a type, forces a variable to only accept assignment of the string literal specified. It's very handy when paired with **type union**.

```
let mouseEvent: 'click' | 'dblclick' | 'mouseup' | 'mousedown';  
mouseEvent = 'click'; // valid  
mouseEvent = 'not a click'; // not valid  
let anotherEvent: mouseEvent; // reusable!
```





# Anatomy of a function

Function name      Parameter type      Return type

```
function echo(message: string): void {  
    console.log(message.toUpperCase());  
}
```

Function body



# Optional parameters

Optional parameters can be specified in TS.

To do so, add a ? postfix to the parameter name.

# Rest parameters

“...” parameters should be noted as an array (use [])





# 5- Classes

Similar to JS with some interesting things:

- ◇ Access modifiers
- ◇ Readonly
- ◇ Getters and Setters






# Access Modifiers

A class can restrict the access to some of their elements using three tags:

- ◇ private
- ◇ protected
- ◇ public (default if not specified)

They can be specified at declaration or in the class' constructor, even though it's not recommended whatsoever.





# *Access modifiers example*

```
class Person {  
    private ssn: string;  
    private firstName: string;  
    private lastName: string;  
  
    // ...  
}
```






# ReadOnly

A class can specify if a variable will be immutable. It's similar to **const**, and it only can only be initialized in:

- ◇ The constructor
- ◇ The own declaration

It is recommended to **only** use it in classes attributes





# Getter and Setter

Use them! The whole Internet (And [Google](#)) tell us to use them with purpose.

Google Style Guidelines says to use them with a **purpose**, not letting them be just a pass-through.





Inheritance, static, abstract

They all work the  
same as JS.








## 6- Interfaces

Interfaces are some sort of specific-use objects in the class sense.

Basically, it provides with a set of attributes and it can be used *as if it were a new type*.

They're really cool, I swear



# But what IS an interface??

Basically, a fancier dictionary.

It helps code be cleaner and set a common ground for classes and functions.

An example shall make this more understandable.





# Function interfaces

We can also use an interface to show *how a function looks like*. What parameters it uses and what type it returns:

```
interface StringFormat {  
    (str: string, isUpper: boolean): string  
}  
  
let format: StringFormat; // here!  
  
format = function (str: string, isUpper: boolean) {  
    return isUpper ? str.toLocaleUpperCase() : str.toLocaleLowerCase();  
};  
  
console.log(format('hi', true)); // HI
```






# Interfaces for classes

A class can *implement* an interface. Similarly to C# and Java, it forms a contract of use among unrelated classes.

It tells classes how to use a set of functions, for example.

Or what parameters they should take into account.

Don't worry an example later on will enlighten you.



# Extending interfaces

Interfaces can extend from other interfaces as objects would do.

```
interface Mailable {  
    send(email: string): boolean  
    queue(email: string): boolean  
}  
  
interface FutureMailable extends Mailable {  
    later(email: string, after: number): boolean  
}
```



# Interfaces extending classes

An interface can extend a **class**.  
However it won't inherit the class' methods or anything, as one would expect from class inheriting.

When an interface extends a class,  
it makes said interface **only  
usable by said class and their  
children**.





## 7- Extra Things

We'll go through several things to bear in mind:

- ◇ Type intersection
- ◇ Guards
- ◇ Type casting
- ◇ Type Assertion
- ◇ Generics
- ◇ Modules





# Type Intersection

When you intersect type A with type B, the result will have both types' properties (all of them).

More interestingly, it also works with interfaces.

Meaning you can build bigger interfaces using smaller ones.







# Guards

Guards are very extensive, and we can't cover them completely.

They can be especially handy to ensure a function returns a certain kind of type to ease conditionals for the compiler.






# Type casting

As in most languages, you can cast types in TS.

You may do so using the *as* keyword, or `<>`.

We recommend the first one, as it makes the code more readable, and the angles are not compatible with some libraries.





# Type assertion

When a function may return a type sometimes or other(s) other times, you can tell the compiler which one you expect.

Use the *as* keyword as before, at the end of the function call.

It also improves readability of your code!






# Generics (a.k.a templates)


Generics are TS' templates, which ease making generic algorithms for various data types.

One may benefit from using generics:

- ◇ Checks types in compiling time
  - ◇ Doesn't need type casting
  - ◇ Allows implementation of algorithms easier
- 



# Generic is very flexible

- ◇ Functions ✓
  - ◇ Classes ✓
  - ◇ Interfaces ✓
- 



# Modules

They are used the same as JS' ES6.

import/export structure.

They work as expected with extensions as well.

You can also export types!





## 8- Conclusions

The same as JavaScript, but better. Fixes many errors, adds hard typing (that's where the *TypeScript* is at), and provides fantastic tools to make some really solid code.

Finally, I don't have to check argument types on every single function.



# References

- ◇ [TS slides](#)
- ◇ [Complete Tutorial \(we used this heavily\)](#)
- ◇ [Exercism track](#)
- ◇ [Official Web](#)
- ◇ [Official TS programmers](#)
- ◇ [Differences between TS and JS](#)

