TypeScript

Introduction to the basics

by Michiel Bouw

About me

Michiel Bouw

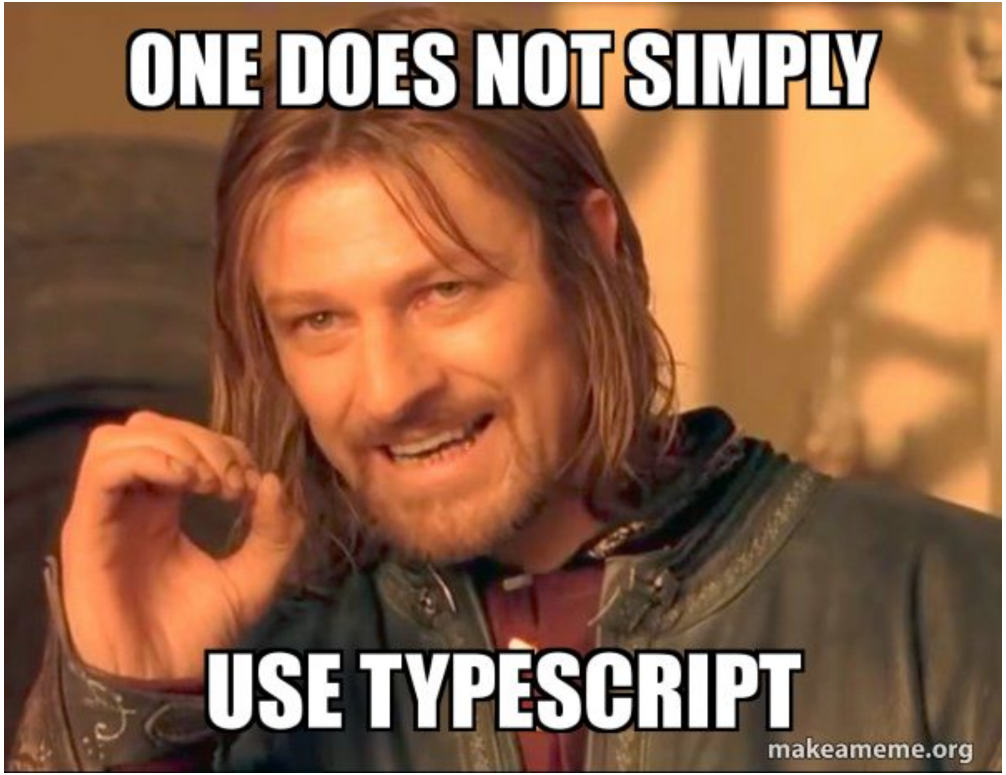
Senior Front-end Developer

Currently working at Smartly.io
6+ yr. Front-end Engineering
3+ yr. Consultancy

https://linkedin.com/in/michiel-bouw

https://michielbouw.nl

Why TypeScript?



Source: https://makeameme.org/meme/one-does-not-6ef0944d8e

What is wrong here?

```
const nameToUpperCase = name => name.toUpperCase();

nameToUpperCase(2020);
```

©2021

TypeScript can see this is wrong by giving hints. It sees that toUpperCase() is a type of string method. We can’t call toUpperCase() on an integer.

Let’s fix this by typing it correctly:

TypeScript can see this is wrong by giving hints. It sees that **toUpperCase()** is a type of string method. We can’t call **toUpperCase()** on an integer.

Let’s fix this by typing it correctly:

```
const nameToUpperCase = (name: string) => name.toUpperCase();

nameToUpperCase(2020);
```

©2021

TypeScript can see this is wrong by giving hints. It sees that toUpperCase() is a type of string method. We can’t call toUpperCase() on an integer.

Let’s fix this by typing it correctly:

```
const nameToUpperCase = (name: string) => name.toUpperCase();

nameToUpperCase(2020); // Error: Type 'number' is not assignable to type 'string’.
```

©2021

TypeScript can see this is wrong by giving hints. It sees that toUpperCase() is a type of string method. We can’t call toUpperCase() on an integer.

Let’s fix this by typing it correctly:

```
const nameToUpperCase = (name: string) => name.toUpperCase();

nameToUpperCase(2020); // Error: Type 'number' is not assignable to type 'string’.

nameToUpperCase('Welcome!'); // No errors
```

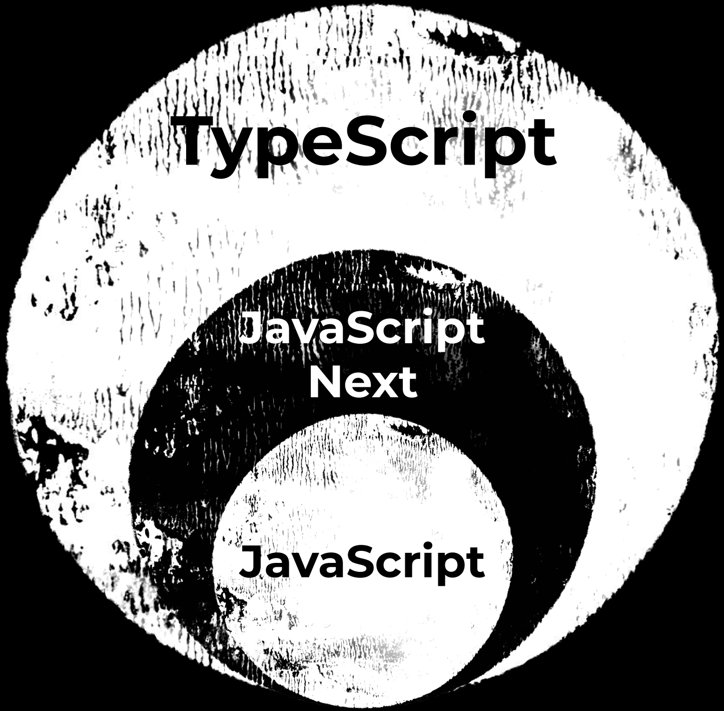
💪 Static type checking is powerful!

What is TypeScript?

TypeScript is an **open source development language** developed by Microsoft and founded in 2012 by a C# architect.

TypeScript is a **superset of JavaScript** and it extends it by adding **static typing** and structuring.

Browsers can’t run TypeScript but it is **transpiled into perfect JavaScript**.



Let’s dive into it!

Why should you use it?

TypeScript ...

1
catches mistakes in your code earlier on

2
supports libraries and API docs

3
structures your code

4
easier to maintain

5
makes it easier to use frameworks

©2021

JavaScript has 6 principle types:

1. string
2. number
3. undefined
4. null
5. symbol
6. boolean

The rest are called **objects.**

TypeScript has:

JavaScript Types *(string, number, undefined, null, symbol, boolean, object)*

+ a couple of extra types

©2021

TypeScript extra types:

1. void
2. tuple
3. enum
4. null, undefined
5. never
6. any

©2021

1/6 void

void - The absence of having any type at all.

In TypeScript, you have to define the return type in your functions. But there are functions which don’t have a return statement, here void comes into play.

1/6 void

To define the return type you can add : + type after the function parameters in brackets (..), like this:

```
const nameToUpperCase = (name: string): string => {
  return name.toUpperCase();
}

// OR shorter

const nameToUpperCase = (name: string): string => name.toUpperCase();
```

What if we don’t return anything:

```
const nameToUpperCase = (name: string): string => name.toUpperCase();

const logNameToUpperCase = (name: string): string => {
  nameToUpperCase(2020);
} // Error: A function whose declared type is neither 'void' nor 'any' must return a
value.ts(2355)
```

1/6 void

In this case we need to use the void type to explicitly say that the function does not return anything:

```
const nameToUpperCase = (name: string): string => name.toUpperCase();

const logNameToUpperCase = (name: string): void => {
  nameToUpperCase(2020);
}
```

2/6 tuple

2/6 tuple

tuple - Organised arrays with predefined types per index. This array has a specific order, but the size is not fixed and it can have different types.

```
const normalMessyArray = ['what', 1, 2, 'this', undefined];

const tuple: [string, number, string, string] =
  ['Show me', 26, 'new', 'things'];
```

3/6 enum

enum (enumeration) - A set of constants but with more friendly names given to it that you will use.

```
enum AppStates {
  hasError,
  isLoading,
  isUserLoggedIn,
}
```

©2021

Be careful the enum begin numbering their members starting at 0. You can manually set the values to the enum values if you need more specific results, like:

```
enum Color {Red = 1, Green = 2, Blue = 3}


enum Color {Red = 'r', Green = 'g', Blue = 'b'}
```

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum:

```
enum AppStates {
  hasError,
  isLoading,
  isUserLoggedIn,
}

let appState: string = AppStates[1]; // results in 'isLoading' > value 2 above
```

4/6 null, undefined

null, undefined - By default in JavaScript null and undefined are subtypes of all other types. You can assign it to something like number.

- null (could be assigned) - the absence of any value
- undefined (has not been assigned) - a variable has been declared but has not yet been assigned a value

©2021

4/6 null, undefined

Not much going on when null or undefined are used on their own:

```
let myUndefinedVariable: undefined = undefined;

let myNullVariable: null = null;
```

In TypeScript null and undefined become different and that may become very useful:

```
typeof null;        // 'object'
typeof undefined;   // 'undefined'

null == undefined;  // true
null === undefined; // false
```

In TypeScript null and undefined become different and that may become very useful:

```
const foo = (name: string | null | undefined) => {
  if (name != null) {
    // name must be a string as `!=` rules out both null and undefined
    ...
  }
  ...
}
```

5/6 never

5/6 never

never - Represents the type of values that never occur (mostly used in error handling).

We always expect the data to be fetched from an API for example, but if we can’t fetch it we need to handle the error:

```
const handleError = (message: string): never => {
  throw new Error(message);
}
```

©2021

6/6 any

6/6 any

any - When you don’t know what type you are dealing with, or are in a situation that you cannot explicitly define, you could use the any type.

```
let notSure: any = 4;

notSure = 'maybe a string instead';

notSure = false; // okay, definitely a boolean
```

©2021

Source: https://knowyourmeme.com/memes/we-need-to-go-deeper

TypeScript extra’s worth mentioning:

1. Type assertions
2. Literal types
3. Template literal types
4. Interfaces
5. Readonly
6. Unknown
7. Errors

©2021

1/7 Type assertions

Type assertions

A way to tell the compiler to trust you.

Yes really!

1/7 Type assertions

Type assertions are useful when you know the type of some entity could be more specific than the current type.

```
let anything: any = 'Am I any or am I string?';
let something = <string>anything; // now it is of type 'string'

// OR with 'as'

const anything: any = 'Am I any or am I string?';
const something = anything as string; // now it is of type 'string'
```

©2021

2/7 Literal types

Literal types

You can use a string literal as a type:

```
let foo: 'Hello';

foo = 'Bar'; // Error: type 'Bar' is not assignable to type 'Hello'.
```

©2021

2/7 Literal types

Combining this in for example a type union it creates a powerful (and useful) abstraction:

```
type Direction =
  | 'Up'
  | 'Down'
  | 'Left'
  | 'Right';

const move = (distance: number, direction: Direction) => { ... }

move(1, 'Up');
move(1, 'Under'); // Error: type 'Under' is not assignable to type Direction.
```

2/7 Literal types

You can also use **boolean** and **number** as literal types, like:

```
type OneToFive = 1 | 2 | 3 | 4 | 5;

type Bools = true | false;

type NumsAndBools = 1 | 2 | false;
```

3/7 Template literal types

NEW IN 2021

Template literal types

NEW IN 2021

Use it to expand on what is already possible with string literal types:

```
type Vertical = 'top' | 'middle' | 'bottom';

type Horizontal = 'left' | 'center' | 'right';

type Alignment = `${Vertical}-${Horizontal}`;
```

3/7 Template literal types

This reduces string repetition and makes code cleaner:

```
type Vertical = 'top' | 'middle' | 'bottom';
type Horizontal = 'left' | 'center' | 'right';
type Alignment = `${Vertical}-${Horizontal}`;

const setAlignment = (alignment: Alignment) => { ... }

setAlignment('top-right');
setAlignment('top-middle'); // Error: type 'top-middle' is not assignable to type Alignment.
```

4/7 Interfaces

Interfaces

Interfaces are a way of naming particular types. It’s basically a group of related methods and properties that describe an object.

©2021

Example usage of interface:

```
interface Person {
  firstName: string;
  lastName: string;
  email: string;
}
```

Properties in an interface can be set optional as well:

```
interface Person {
  firstName: string;
  lastName: string;
  email: string;
  employeeNumber?: number;
}
```

Like classes, interfaces can extend each other. This allows you to reuse and not include double code like this:

```
interface Person {
  firstName: string;
  lastName: string;
  email: string;
}
```

```
interface Employee extends Person {
  employeeNumber: number;
}
```

```
const user: Employee =
  { firstName: 'Michiel', lastName: 'Bouw', email: 'a@b.com' };

// Error: Property 'employeeNumber' is missing in type '{ firstName: string;
          lastName: string; email: string; }' but required in type 'Employee'.ts(2741)
```

©2021

5/7 Readonly

Readonly

Use readonly to prevent errors from mutating objects.

©2021

5/7 Readonly

When an array or a tuple is marked as readonly, TypeScript will throw an error when you try to to add, remove, or update items in those objects. This is especially helpful in functional programming to avoid side effects.

```
string[]

vs.

readonly string[] OR ReadonlyArray<string>
```

©2021

6/7 Unknown

Unknown

Prefer unknown over any to get better type safety.

©2021

With unknown type, TypeScript will correctly surface errors such as calling non existent methods:

```
const anyData: any = getRandomData();

anyData().method(); // No errors!

const unknownData: unknown = getRandomData();

unknownData().method(); // Error: Object is of type 'unknown'.ts(2571)
```

6/7 Unknown

If you as a developer gain more knowledge about the type, you can also use unknown with double assertions to clarify the intent:

```
interface DataWithMethod {
  method: () => void;
}

const unknownData = getRandomData() as unknown as DataWithMethod;

unknownData().method(); // No errors!
```

7/7 Errors

Errors

Although TypeScript tries to make the error messages as helpful as possible, this could lead to an overload of information thrown at you that you might not really understand how to interpret.

©2021

Let’s look a bit closer at the different type of errors to help you understand them.

These are two different types of errors you can get:

- **Succinct**
- **Detailed**

©2021

Succinct - This will provide you an example conventional description of the error with a message:

```
TS2345: Argument of type '{ foo: number; bar: () => string; }' is not
assignable to parameter of type 'SomethingComplex'.
```

©2021

7/7 Errors

Detailed - Most of the time we need some more information on why the error is happening, that’s why there are detailed error messages:

```
[ts]
Argument of type '{ foo: number; bar: () => string; }' is not assignable to
parameter of type 'SomethingComplex'.
  Types of property 'bar' are incompatible.
    Type '() => string' is not assignable to type 'string’.
```

The detailed error will provide you with a chain of things that happened.

The previous one should read like:

```
ERROR: Argument of type '{ foo: number; bar: () => string; }' is not
assignable to parameter of type 'SomethingComplex'.

WHY?
CAUSE ERROR: Types of property 'bar' are incompatible.

WHY?
CAUSE ERROR: Type '() => string' is not assignable to type 'string'.
```

The errors provided by TypeScript should help you fix bugs

You can use the TSXXXX error code in the succinct error to find other similar cases online to help you out if needed.

©2021

Summary

TypeScript is a superset of JavaScript

TypeScript is transpiled into perfect JavaScript

TypeScript catches mistakes in your code early on

💪 Static type checking is powerful!

Costs of TypeScript

1
Setup time, training

2
Typing overhead

3
Recruiting

Benefits of TypeScript

1
Developer tooling

2
API Docs

3
Refactoring effort

4
Type safety, but optional

5
Code readability

©2021

How to get going with TypeScript?

Try to convert any of your existing (JavaScript) application to use TypeScript.

How to use it?

1. Install the npm package of TypeScript, create a **tsconfig.json** and setup the compiler

or

2. **Choose TypeScript at setup** of your favorite framework (most already have this feature)

Useful links

- Official TypeScript docs: https://www.typescriptlang.org/docs/home.html
- TypeScript Deep Dive: https://basarat.gitbook.io/typescript/
- More TypeScript resources: https://github.com/dzharii/awesome-typescript

©2021

Thank you!

Michiel Bouw

https://linkedin.com/in/michiel-bouw

https://michielbouw.nl