



TypeScript

(super/sub/whatever)set of JavaScript

Frontend Junior Program - 2022

Looking for love in all the wrong frameworks



Hype Driven Development

Life on the Bandwagon

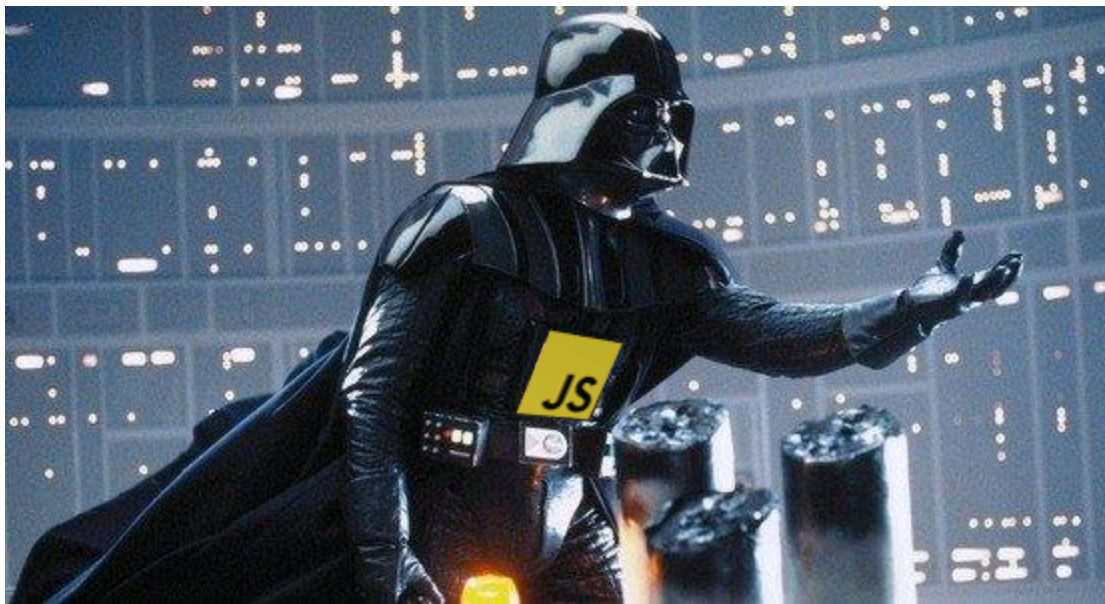
ORLY?

@ThePracticalDev



„This is not the language you think about...”





TypeScript is basically a language extension on JavaScript – just with a bit lot more complexity*: constantly yelling ([error messages](#)), disrespecting the true power of its ancestor ([forcing static types](#)), arrogantly demanding to except its friends ([type definitions](#)), adding infinite list of generational traumas ([changelog](#)) and behavioral problems ([bugs](#)).

** it is not that the ancestor would not lack its fair share of issues, like [this](#).*

TypeScript

The TypeScript (TS) language is almost** backward compatible with JavaScript

But browsers cannot consume directly: the [TypeScript code needs to be compiled*](#) to [JavaScript](#) before deployed to the server.

TypeScript does static type-checking, which means: it [checks the types compile time](#), not run time!

TypeScript does have an excellent [Handbook](#), so if you are interested look no further to learn it.

* actually, it's called transpiling:

source code



compiling

lower-level code
(machine code usually)

source code



transpiling

code on similar
abstraction level
(another source code)

Usually, it is done by the TypeScript transpiler, however, [Babel](#) also can transpile TypeScript code.

** not all valid JavaScript code is valid TypeScript code, you will see...



why would you missed that?

Is it worth to **learn TypeScript**?

In Front-End development, **you have no choice**: it is a must. It is widely used in projects, and it teaches (forces) you planning, to think on a higher abstraction level.

But what are the features that renders it so important?

Static type-checking

You may ask, why type-checking is necessary? *“I do know when a variable is a string or a number - I don’t need a tool for that!”*

Sure, you do, but the data structures we use are extremely complex sometimes. It is really impossible to track all the types of every [properties of a deeply nested object](#).

IntelliSense

As TypeScript knows the expected types and interfaces, with a proper IDE support it let’s you focus on what is important.

Supporting [new JavaScript features](#) + [enums](#)

The browsers’ support always lags behind and inconsistent between browsers - not to mention oldest browsers.

While it is not crucial - or makes sense - to use every language novelties immediately, it was an incredible moment, when TS started to support [optional chaining](#).

THE GALACTIC REPUBLIC's new *Venator*-class Star Destroyer is fast enough to chase down blockade runners and big enough to lead independent missions such as the liberation of Utapau. A flotilla of these medium-weight, versatile multi-role warships can blast through the shields of a Trade Federation battleship with ease. The hangars of the *Venator*-class are much larger than older Star Destroyers like the *Victory*-class, and can support hundreds of fighters. The ship is also capable of planetary landings as a military transport and can be an escort for battleships in the Republic armada. However, the primary function of the *Venator*-class is its role as a fighting ship and starfighter carrier, making it a firm favorite with Jedi fighter aces.

DATA FILE
Manufacturer: Kumb Drive Yards
Make: Versorix class 5.137 motor
Dimensions: length 1,137 m (3,729 ft); wingspan 548 m (1,797 ft); height (in flight) 268 m (879 ft)
Max. acceleration (linear, in open space): 3,000G
Hyperdrive: class 10; 60,000 light year effective range
Crew: 7,400
Armament: 8 heavy turbolaser turrets; 2 medium dual turbolaser cannons; 52 point defense laser cannons; 4 proton torpeda tubes; 6 tractor beams
Complement: 192 V-wing fighters; 192 Eta-2 Interceptors; 36 ABC-17 fighters; 24 military walkers; 40 LAATs (Low Altitude Assault Transport) infantry dropships; miscellaneous shuttles

Winter typically lasts 25 ABC-170 nights.

Transportation) group

Small box

BeCres-770



100

protect the ship's loading **Now that's**

Moll und

even, and

but that i

... ..

everything.

— 1997 —

TypoScript

Introduction

2. EDAM Systems

CARRIER ROLE

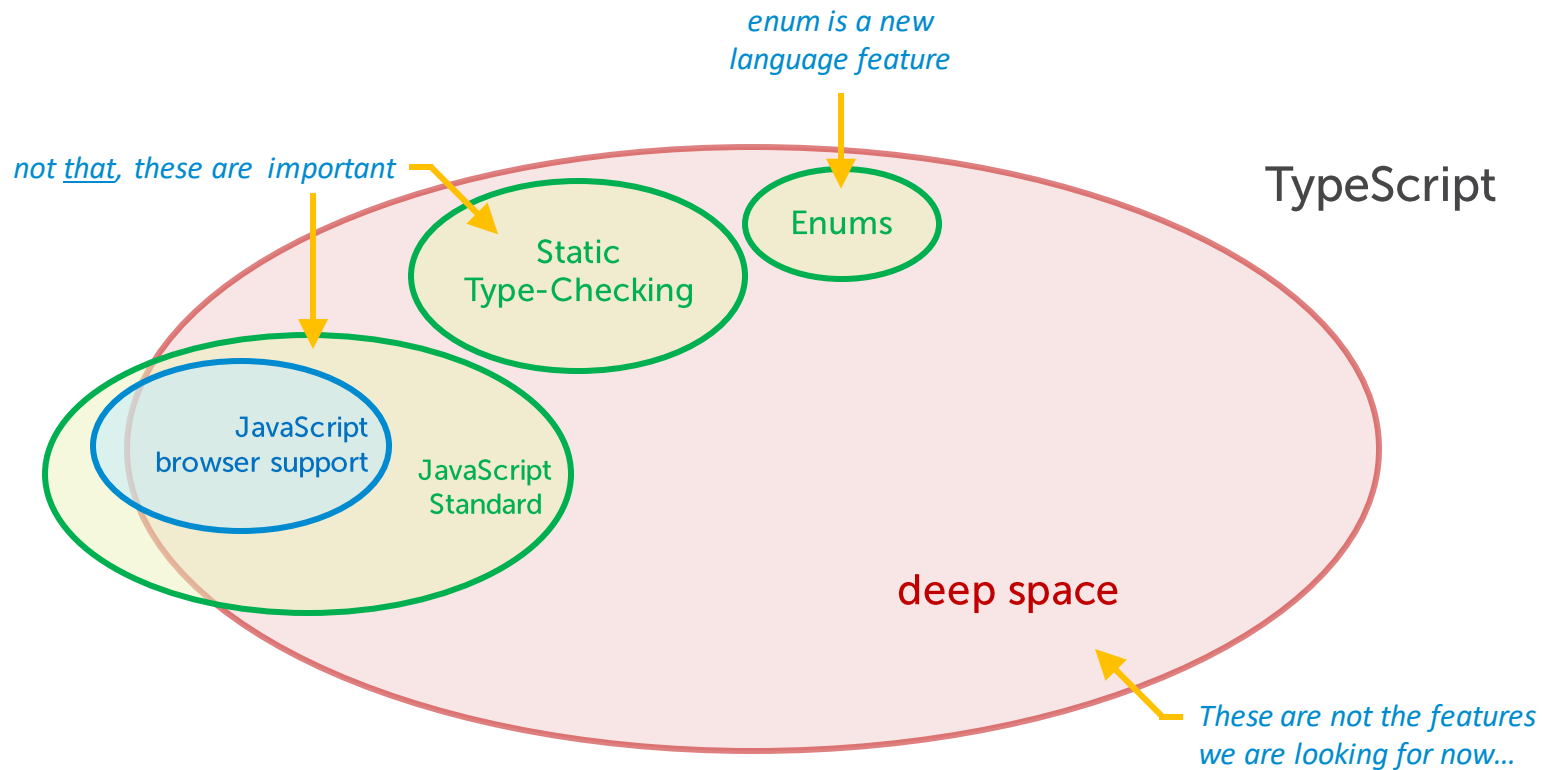
The long dorsal flight deck of the *Venator* class enables hundreds of starfighters to launch rapidly. The slow opening and closing of the deck's armored bow doors, however, can leave the vessel vulnerable. This weakness is compensated for by strong deflector shielding around the deck's entrance, but the design flaw will be eliminated in future *Star Destroyers*.

Heavy-Duty Firearm

A *Vesuvius*-class Star Destroyer's eight DBY-327 heavy turbolaser turrets are the standard requirement in naval gunnery for intense inter-ship combat and planetary bombardment. The DBY-327's precise, long-range tracking mode enables it to hit a target vessel at distances of over ten light minutes, while the turret can rotate in three seconds in its climb-fighting, fast-tracking mode. Seven different blast intensities provide a choice between crippling shots and outright vaporization of the enemy. The *Vesuvius*-class, as a true warship, can feed almost its entire reactor output to its heavy guns when required.

THE KUAT LEGACY

Kint Drive Yards, the manufacturer of the *Wrestler*, claim the Republic is winning the Clone Wars with this ship and their other powerful, wedge-shaped vessels. But the construction of *Wrestler*-class vessels is already slowing in favor of more robust, mile-long Imperial class (transmed *Imperial* class after the Jedi Purge) and hangarless *Tecan* class Star Destroyers. These ships will see service for decades to come, as the Republic is transformed into the Empire. The Imperial Starfleet will justify its existence in treading war against Separatist hordes, dissident nobles and even, it is rumored, deterring barbarian invaders from outside the galaxy



Playground

There is a [playground](#) for TypeScript, where you can try out all the features online.

barely valid JS code, however,
TS still transpiles it

this is more interesting: it is a valid JS
code, still, TS does not like it

The screenshot shows the TypeScript Playground interface. The top navigation bar includes 'TS TypeScript', 'Download', 'Docs', 'Handbook', 'Community', 'Tools', and a 'Search Docs' search bar. Below this is a secondary bar with 'Playground', 'TS Config', 'Examples', 'What's New', and 'Settings'. The main area is divided into two panes. The left pane, titled 'v4.2.3', 'Run', 'Export', and 'Share', contains the following TypeScript code:

```
1  const object = {};  
2  
3  A long time ago, in a galaxy far, far away...  
4  
5  It is a period of civil war. Rebel  
6  spaceships, striking from a hidden  
7  base, have won their first victory  
8  against the evil Galactic Empire.  
9  
10 During the battle, Rebel spies managed  
11 to steal secret plans to the Empire's  
12 ultimate weapon, the Death Star, an  
13 armored space station with enough  
14 power to destroy an entire planet.  
15  
16 Pursued by the Empire's sinister agents,  
17 Princess Leia races home aboard her  
18 starship, custodian of the stolen plans  
19 that can save her people and restore  
20 freedom to the galaxy...  
21  
22 object.prop = "property";
```

The right pane, titled '.JS', '.D.TS', 'Errors' (with a red badge showing '174'), 'Logs', and 'Plugins', shows the transpiled JavaScript code:

```
"use strict";  
const object = {};  
A;  
long;  
time;  
ago, in a;  
galaxy;  
far, far;  
away;  
It;  
is;  
a;  
period;  
of;  
civil;  
war.Rebel;  
spaceships, striking;  
from;  
a;  
...
```

Basic Syntax

explicit* type annotation



```
const str: string = "Skywalker";
```

* a very important feature of TypeScript, that we don't need declare types explicitly in every case: TS is smart, when it is possible, [it can figure out and check the types without any annotations](#).

Primitives

while TS is not a linter, it could be very nitpicking on JS code



```
const yoda = {};  
yoda.property = "wise";
```

now, TS is happy with **any**



```
const luke: any = {};  
luke.property = "young";
```

primitives



```
const number: number = NaN;  
const string: string = "Skywalker";  
const boolean: boolean = Boolean();  
const _undefined: undefined = undefined;  
const _null: null = null;
```

bonus: object
(object is not a primitive)



```
const object: object = {};
```

```
const array: array
```

[?] Array interface Array<T>
[?] ArrayBuffer
•○ ArrayBufferConstructor
🔑 ArrayBufferLike
•○ ArrayBufferTypes
•○ ArrayBufferView

Errors in code

Property 'property' does not exist on type '{}'.
Cannot find name 'a'.

'const' declarations must be initialized.

Exported variable 'array' has or is using private name 'a'.

there is no such a type in JS like **array**, we already told you

Basics

both can work for **arrays**
(depending on your config, sometimes
it is permitted only one of these)



```
const arraySimple: string[] = [];  
const arrayGenerics: Array<string> = [];
```

any can be really anything



```
{  
  let anything: any;  
  anything = NaN;  
  anything = "Skywalker";  
  anything = Boolean();  
  anything = {};  
}
```

TS **recognizes** that *yoda* is a string



```
let yoda = "Wise";  
yoda = Infinity;
```

type annotation on a **parameter**
and on a **return type**



```
function doOrDoNot(name: string): string {  
  | return name;  
}
```

```
doOrDoNot(yoda);
```

any will fit for a string as well



```
doOrDoNot(anything);
```

but a **number** does not *count*



```
doOrDoNot(NaN);
```

vice versa: a string won't fit for a
number type



```
let number: number = doOrDoNot(yoda);
```

Errors in code

- Type 'number' is not assignable to type 'string'.
- Argument of type 'string[]' is not assignable to parameter of type 'string'.
- Type 'string' is not assignable to type 'number'.

Inferred types

TS will know the return type →

```
function iAmYourFather(name: string) {  
  return name;  
}
```

with default parameter as well →

```
function lackOfFaith(name = "Motti") {  
  return name;  
}
```

inferring types is a life saver:
nested and chained functional code
would be insanely unreadable,
if we were forced to add type
annotations to everywhere

```
number = lackOfFaith();  
  
["Vader", "Luke"].map( e => e.find());
```

→ this is why TypeScript needs **type definitions**: TS knows that there is no such a method, like *find* on a string

Errors in code

Type 'string' is not assignable to type 'number'.

Type 'string' is not assignable to type 'number'.

Property 'find' does not exist on type 'string'.


```
const skywalkers =
  ["Vader", "Luke"].map((e = "Leia") =>
    e?.trim() ?? "DefinitelyNotRey");

const weapon = {
  lightSaber: {
    color: function() {
      return "red";
    }
  }
};

weapon?.lightSaber;
weapon?.lightSaber?.color?.();
weapon ?? {};

[...skywalkers];
({...weapon});
```



```
"use strict";
var __assign = (this && this.__assign) || function () {
  __assign = Object.assign || function(t) {
    for (var s, i = 1, n = arguments.length; i < n; i++) {
      s = arguments[i];
      for (var p in s) if (Object.prototype.hasOwnProperty.call(s, p))
        t[p] = s[p];
    }
    return t;
  };
  return __assign.apply(this, arguments);
};

var __spreadArray = (this && this.__spreadArray) || function (to, from) {
  for (var i = 0, il = from.length, j = to.length; i < il; i++, j++)
    to[j] = from[i];
  return to;
};

var _a, _b;
var skywalkers = ["Vader", "Luke"].map(function (e) {
  var _a;
  if (e === void 0) { e = "Leia"; }
  return (_a = e === null || e === void 0 ? void 0 : e.trim()) !== null && _a !== void 0 ? _a : "DefinitelyNotRey";
});
var weapon = {
  lightSaber: {
    color: function () {
      return "red";
    }
  }
};

weapon === null || weapon === void 0 ? void 0 : weapon.lightSaber;
(_b = (_a = weapon === null || weapon === void 0 ? void 0 : weapon.lightSaber) === null || _a === void 0 ? void 0 : _a.color) !== null && weapon !== void 0 ? weapon : {};
__spreadArray([], skywalkers);
(__assign({}, weapon));
```

ESNext support

Depending on the configuration,
TS could be very helpful for old
browsers.

Type Alias vs Interface

We can create own types...

...in 2 different ways. `~_(\ツ)_/~`

there are **type aliases** →

```
type Name = string;

interface Person {
  name: Name;
}

type LukeString = Person & {
  age: number;
}
```

and **interfaces** →

```
interface YodaString extends Person {
  age: number;
}
```

```
let luke: LukeString = {
  name: "Luke",
  age: 19
}
```

```
let yoda: YodaString = {
  name: "Yoda",
  age: 900
}
```

```
luke = yoda;
```

TS is smart: the **structure** matters only →

type-checking is compile-type!

there are no types in
the transpiled code



```
"use strict";
var luke = {
  name: "Luke",
  age: 19
};
var yoda = {
  name: "Yoda",
  age: 900
};
luke = yoda;
```

*"**Type aliases and interfaces** are very similar, and in many cases you can choose between them freely."*

*Almost all features of an interface are available in type, the key distinction is that a **type cannot be re-opened to add new properties** vs an **interface which is always extendable**."*

Enum

Enums are **enumerables**

They represent well defined, distinct values.

numeric enum →

```
enum Jedi {  
  Luke,  
  Yoda  
}
```

enums are referred by the dot notation →

```
const jedi: Jedi = Jedi.Luke;
```

no error. well... →

```
const yoda: Jedi = NaN;
```

string enum →

```
enum Skywalker {  
  Luke = "LUKE",  
  Leia = "LEIA"  
}
```

```
const skywalker: Skywalker = Skywalker.Luke;
```

error. well... →

```
const luke: Skywalker = "LUKE";
```

*please decipher **what is going on here**, it is very revealing*



```
"use strict";  
var Jedi;  
(function (Jedi) {  
  Jedi[Jedi["Luke"] = 0] = "Luke";  
  Jedi[Jedi["Yoda"] = 1] = "Yoda";  
})(Jedi || (Jedi = {}));  
var jedi = Jedi.Luke;  
var yoda = NaN;  
var Skywalker;  
(function (Skywalker) {  
  Skywalker["Luke"] = "LUKE";  
  Skywalker["Leia"] = "LEIA";  
})(Skywalker || (Skywalker = {}));  
var skywalker = Skywalker.Luke;  
var luke = "LUKE";
```

Q&A