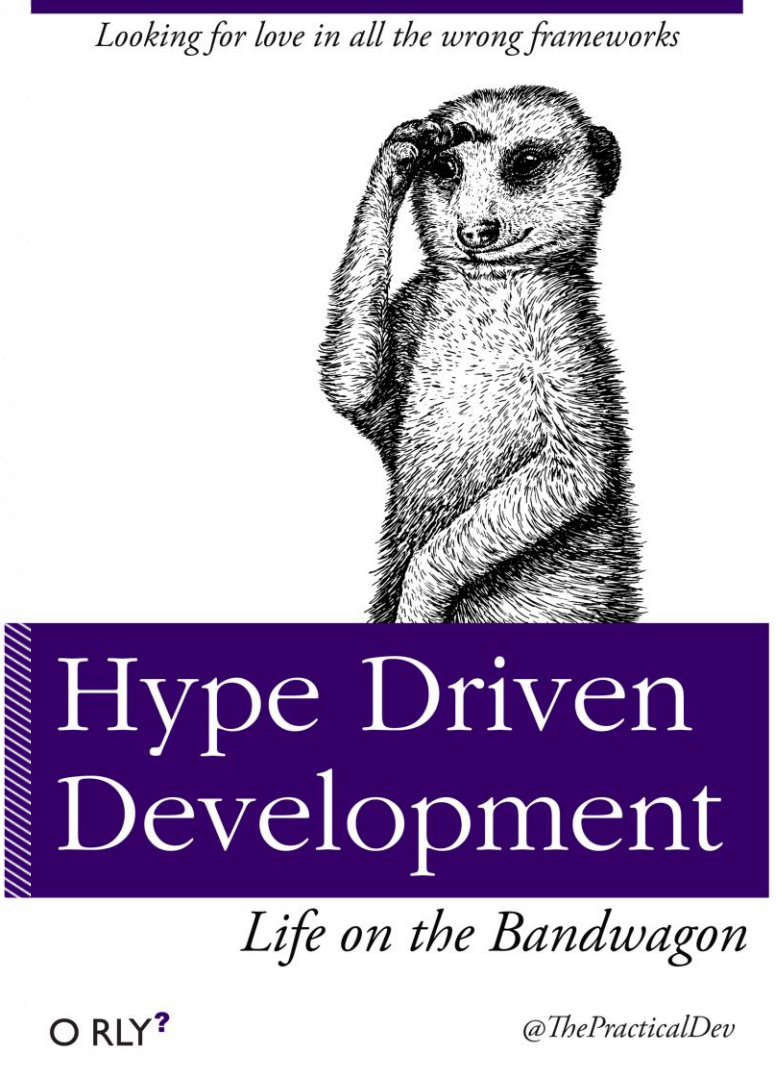
TypeScript

(super|sub|whatever)set of JavaScript

Frontend Junior Program - 2022



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

Microsoft

[JavaScript - TypeScript: Making .NET Developers Comfortable with JavaScript | Microsoft Docs]

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.*

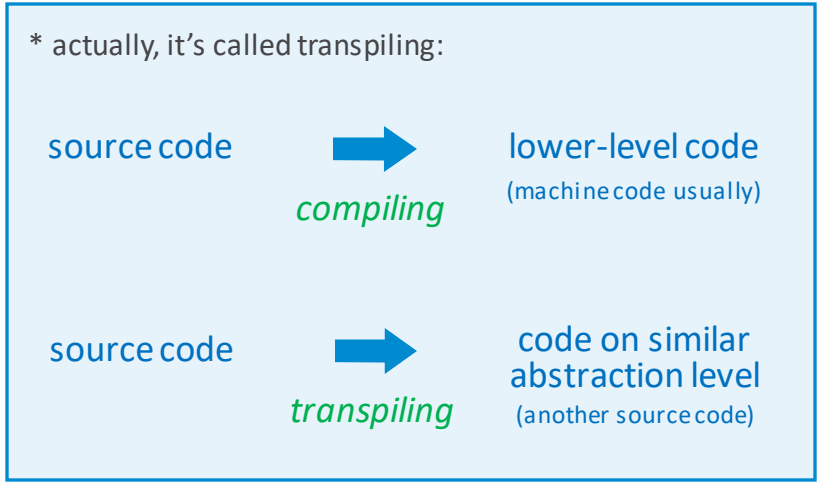
TypeScript

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.



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.

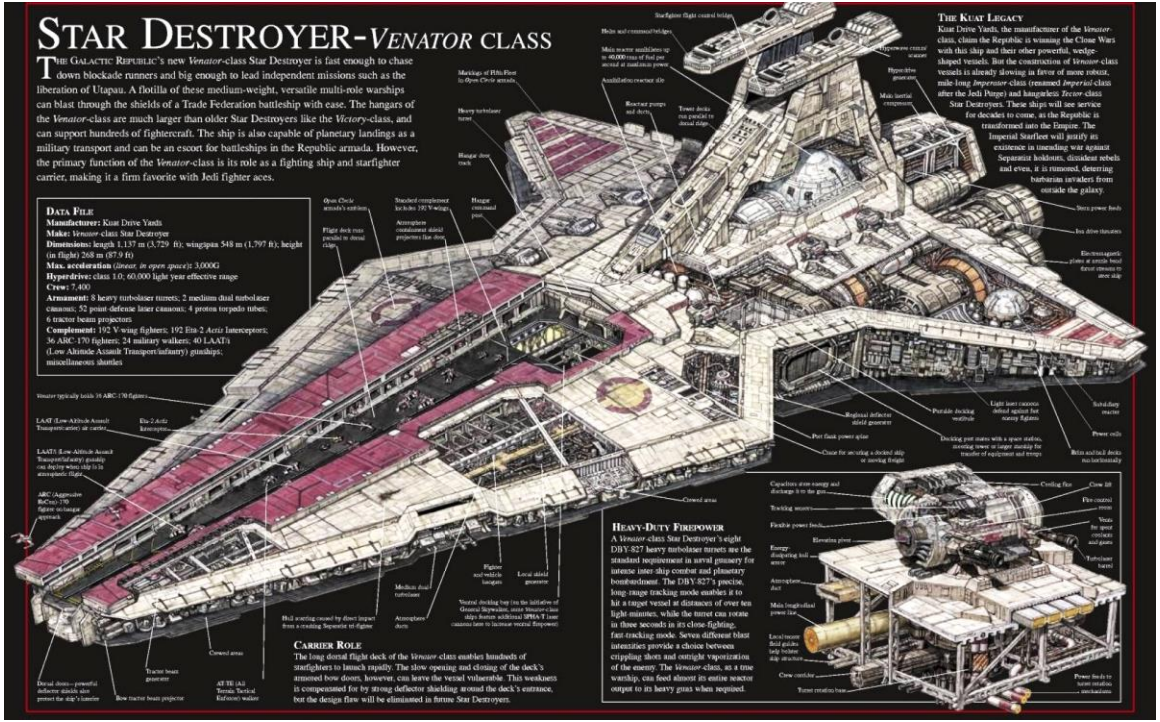
IntelliSense

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

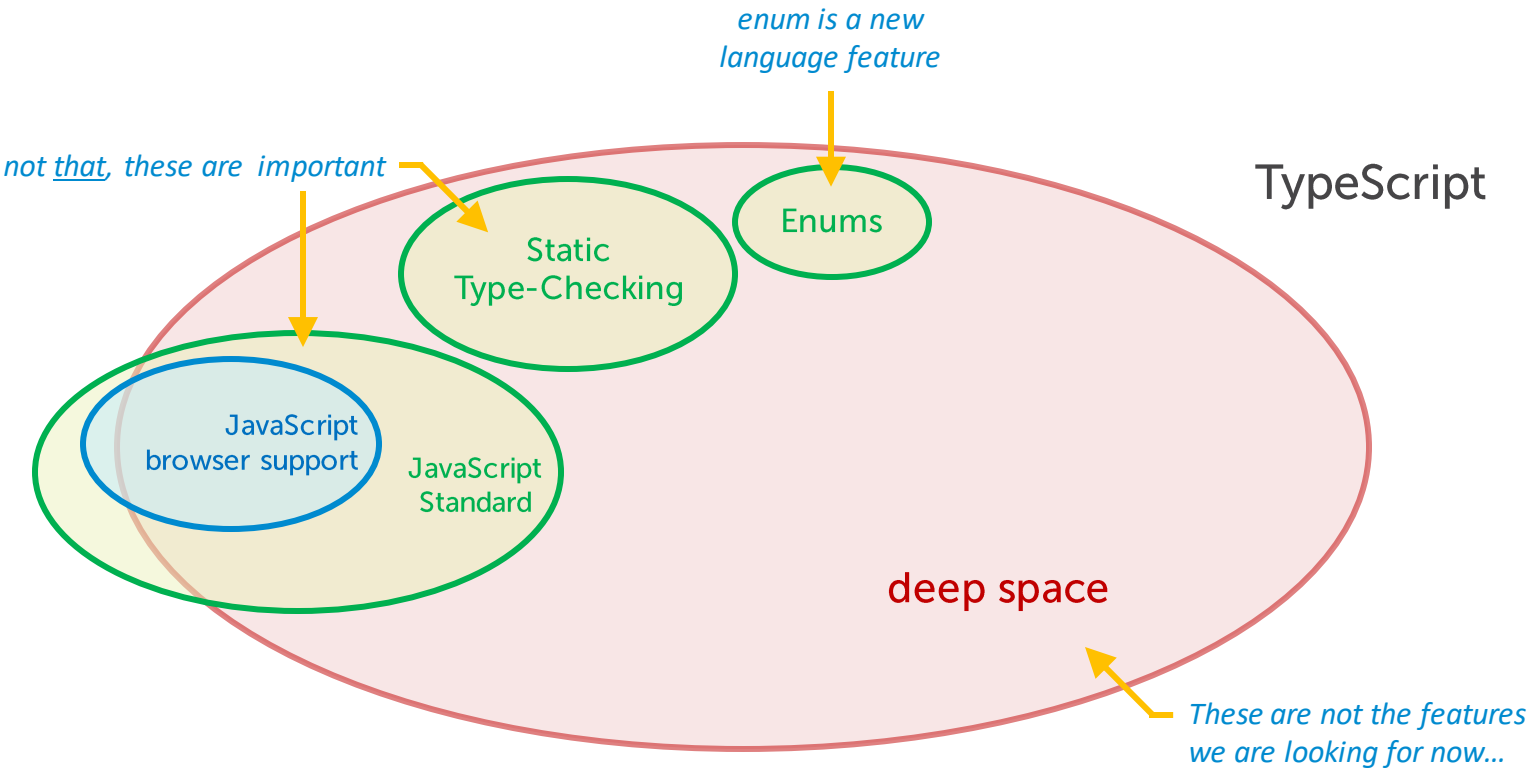
Supporting new JavaScript features + enums

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.



Well, undoubtedly, the full feature set could be shockingly complex at the first sight, but that is just because it is *shockingly complex*. Luckily, you hardly will use everything, and learning the important parts is relatively easy. You can be confident in TypeScript with just mastering the fundamentals – that is important in projects.

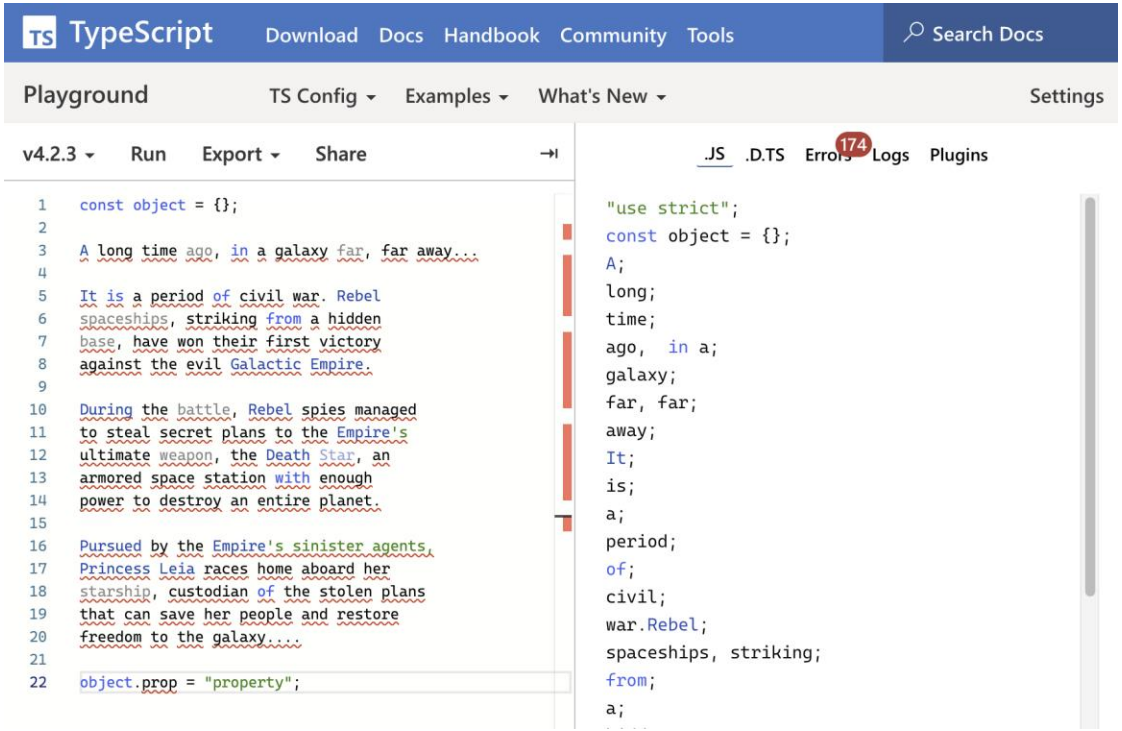

enum is a new
language feature
not that, these are important
TypeScript
Static
Type-Checking
Enums
JavaScript
browser support
JavaScript
Standard
deep space
These are not the features
we are looking for now…

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



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

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

Errors in code

> Property 'property' does not exist on type '{}'.

> 'const' declarations must be initialized.

> Cannot find name 'a'.

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

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;
}

let number: number = iAmYourFather("Luke");
```

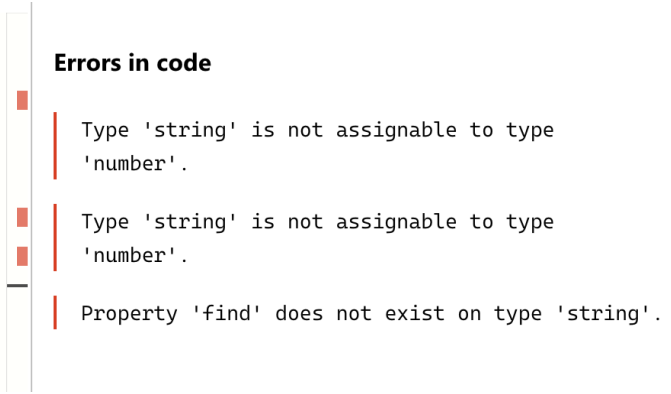
with default parameter as well

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

number = lackOfFaith();
```

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

```
["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

```
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.co
weapon !== 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

and interfaces

TS is smart: the structure matters only

```ts
type Name = string;

interface Person {
    name: Name;
}

type LukeString = Person & {
    age: number;
}

interface YodaString extends Person {
    age: number;
}

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

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

luke = yoda;
```

type-checking is compile-type!

there are no types in the transpiled code

```ts
"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";
```

edu_hu@epam.com