scan QR code for slide-deck and source-code

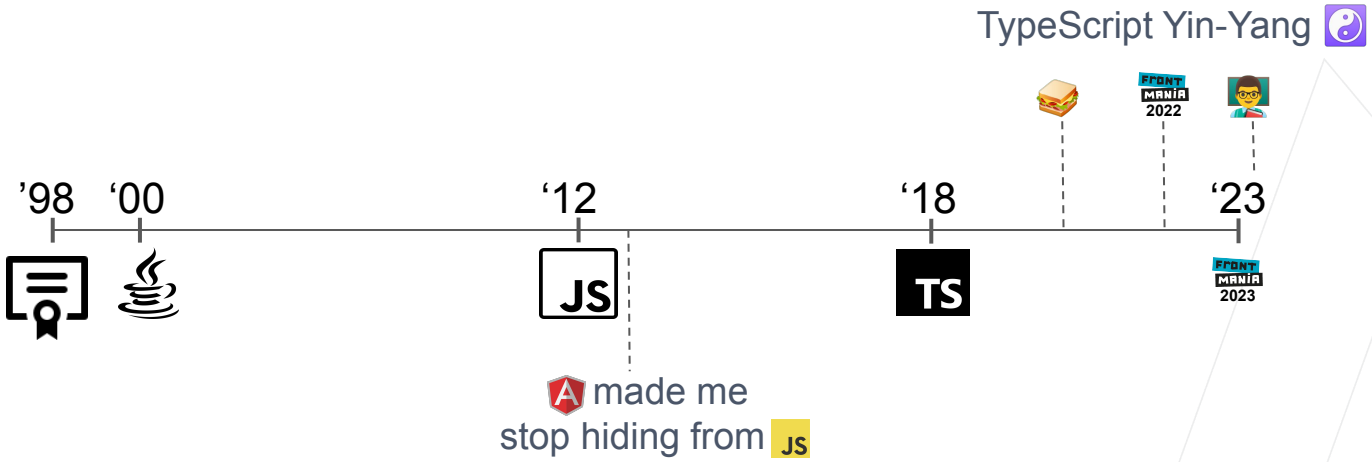
TypeScript inaccuracies?

Let's fix those types!

Emil van Galen

divotion

About myself …



@EmilVanGalen

@EmilVanGalen@fosstodon.org

Agenda

- TypeScript Inaccuracies (and its flavours)
- The `ts-reset` library (& declaration merging)
- (More) accurate types
- Type Programming
- Live coding
- Coding conventions & recommendations

Before I (really) start …

scan QR code for slide-deck and source-code

note that:

- probably no time left for questions 😞
- but on GitHub you will find:
  - this slide-deck
  - live-coding sources (incl. in-between steps)
- scratching the surface of what's possible
- it's totally fine, when you're a bit overwhelmed

TypeScript Inaccuracies

occur when type is:

- **too** broad (given the situation)
- **too** lenient (not fully type-safe)
- incorrect

Too broad types

```
const mapIsoDate = (isoDate: string | null) => {
  if (!(typeof isoDate === 'string')) {
    return null;
  }

  return new Date(isoDate);
};
```

Too broad types

```
const mapIsoDate = (isoDate: string | null) => {
  // ..
}
```

inferred return type is too broad

```
const dateTypeExpected = mapIsoDate('2023-09-01');
//    ^? const dateTypeExpected: Date | null
```

Two-slash `^?` comment shows info normally shown on hover

```
const nullTypeExpected = mapIsoDate(null);
//    ^? const nullTypeExpected: Date | null
```

Too broad types

```ts
const isInThePast = (isoDate: string) => {
  const date = mapIsoDate(isoDate);

  return date.getTime() < Date.now();
  // TS18047: 'date' is possibly 'null'.
}

const mapIsoDate = (isoDate: string | null) => {
  // ..
}
```

Too broad types

```
const isInThePast = (isoDate: string) => {
  const date = mapIsoDate(isoDate);

  return date!.getTime() < Date.now();

}

const mapIsoDate = (isoDate: string | null) => {
  // ..
}
```

using a non-null assertion in **production code** 😨

inferred return type is too broad

Too broad types

```ts
const isInThePast = (isoDate: string) => {
  const date = mapIsoDate(isoDate);

  return date ? date.getTime() < Date.now() : false;
  // FIXME: date should 'automagically' be non-null 🤔
}

const mapIsoDate = (isoDate: string | null) => {
  // ..
}
```

Too lenient types

are typically due to the any type

Too lenient types

the any type is problematic:

- since it ***disables*** type checking
- because it's **leaking**: tkdodo.eu/blog/beware-the-leaking-any
- as there's a modern type-safe alternative
  - the unknown type *

* read more about this here: https://mariusschulz.com/blog/the-unknown-type-in-typescript

Too lenient types

`any` can unexpectedly occur in (built-in) typings

```
interface JSON {
  // ...
  parse(
    text: string,
    reviver?: (this: any, key: string, value: any) => any,
  ): any;
  // ...
}
```

* the `parse` method could alternatively be ‘monkey-patched’ by “ts-reset” lib to return `unknown` instead

Too lenient types

`any` can be spotted with `type-coverage` CLI tool

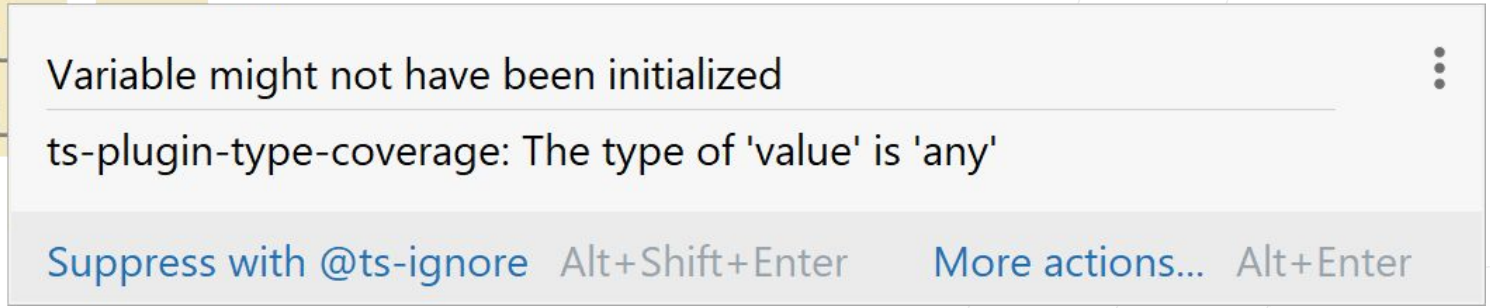
```ts
let value: any;

value.trim();
value('hello');
value.foo;
```

```
$ type-coverage --detail .
../value-any.ts:1:5: value
../value-any.ts:3:1: value
../value-any.ts:3:7: trim
../value-any.ts:4:1: value
../value-any.ts:5:1: value
../value-any.ts:5:7: foo
```

Too lenient types

`any` can be spotted with its Language Service plugin

```
let value: any;

value.trim();
val
val
```



* screenshot is from WebStorm with `ts-plugin-type-coverage` plugin configured in `tsconfig.json` file

Too lenient types

`any` is reported by `typescript-coverage-report`

TypeScript coverage report

Summary

Percent	Threshold	Total	Covered	Uncovered
89.25%	80%	186	166	20

Files

Filename	Percent	Total	Covered	Uncovered
src\any-to-unknown.solution.ts	100.00%	45	45	0
src\any-to-unknown.ts	54.17%	24	13	11
src\foo-from-json.solution.ts	100.00%	17	17	0
src\foo-from-json.ts	76.92%	13	10	3
src\union.solution.ts	100.00%	39	39	0
src\union.ts	100.00%	42	42	0
src\value-any.ts	0.00%	6	0	6

Incorrect types

typically occur in typings that are :

- separate, `@types/`.. package (DefinitivelyTyped)

Incorrect types

lead to:

- confusion
- wasting time
- unwanted type assertion usage

Using `ts-reset` library

you can fix some annoyances in built-in typings

```
const result = JSON.parse("{}");
//    ^?const result: any
```

Using `ts-reset` library

you can fix some annoyances in built-in typings

```ts
import "@total-typescript/ts-reset";

const result = JSON.parse("{}");
//    ^?const result: unknown
```

Using `ts-reset` library

you can fix some annoyances in built-in typings

```
const filteredArray = [1, 2, undefined].filter(Boolean);
//    ^?const filteredArray: (number | undefined)[]
```

Using `ts-reset` library

you can fix some annoyances in built-in typings

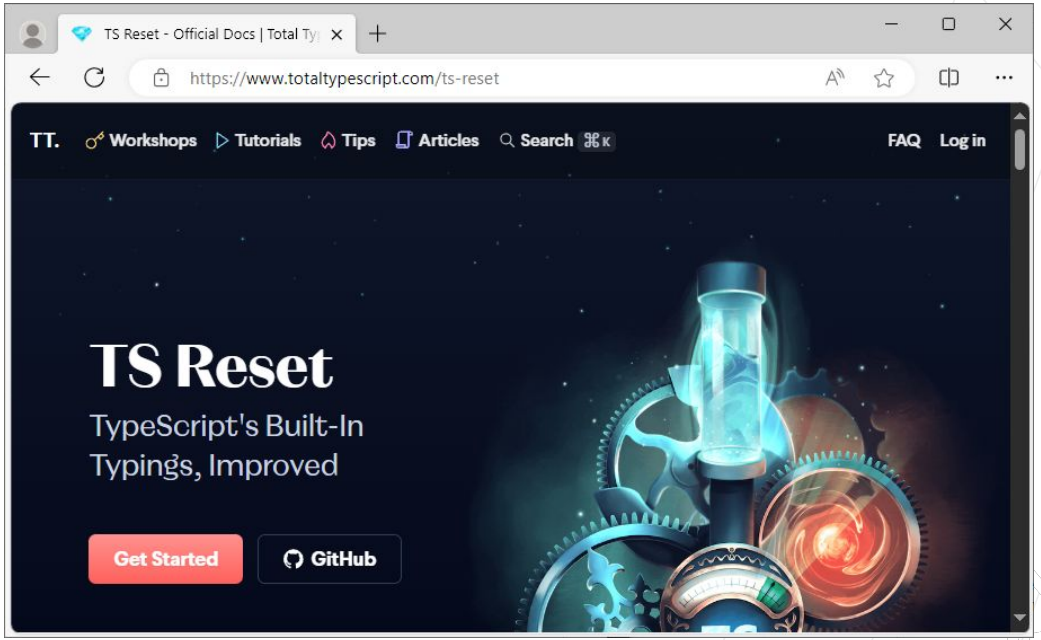
```ts
import "@total-typescript/ts-reset";

const filteredArray = [1, 2, undefined].filter(Boolean);
//    ^? const filteredArray: number[]
```

Using `ts-reset` library

is only advised for applications

Using `ts-reset` library

is further described on: totaltypescript.com/ts-reset



Declaration merging

allows `ts-reset` to ‘override’ build-in typings

Declaration merging

Live-coding with

Twoslash Query Comments

Orta | 30,152 installs | ★★★★★ (9)

Adds support for twoslash query comments (// ^?) in TypeScript and JavaScript projects

expect-type

Compile-time tests for types. Useful to make sure types don't regress into being overly-permissive as changes go in over time.

* live-coding source-code can be found in `1..-declaration-merging.ts` files in GitHub repository

Declaration merging

is described in more detail in TypeScript Handbook:
typescriptlang.org/docs/handbook/declaration-merging.html

(More) Accurate types

could be achieved using:

- declaration merging ✔
- ➢ generics and / or `keyof` keyword

Property getter

that’s is generic

```ts
const getProperty = <Type extends object>(
  obj: Type, key: keyof Type
) => obj[key];


const x = getProperty({ x: 1, y: 2}, 'x');
//    ^? const x: number
```

keyof keyword produces union of all keys

Property getter

that’s is generic

```
const getProperty = <Type extends object>(
  obj: Type, key: keyof Type
) => obj[key];


const x = getProperty({ x: 1, y: '2'}, 'x');
//    ^? const x: number | string
```

there’s actually a bug in the return type

Property getter

that's is generic… must also have type parameter for `Key`

```ts
const getProperty = <Type extends object, Key extends keyof Type>(
  obj: Type, key: Key
) => obj[key];


const x = getProperty({ x: 1, y: '2'}, 'x');
//    ^? const x: number
const y = getProperty({ x: 1, y: '2'}, 'y');
//    ^? const y: string
```

Property getter

could also support a **nested** property path

```
const obj = { a: { nested: { property: 'hello' } } };

const _ = getNestedProperty(obj, 'a.nested.property');
//    ^? const _: string
```

(More) Accurate types

could be achieved using:

- declaration merging ✔
- generics and / or `keyof` keyword ✔
- ➢ function overloading

Function overloading

allows a different return type for different argument type *

```
function mapIsoDate(isoDate: string      ): Date;
function mapIsoDate(isoDate: null        ): null;
function mapIsoDate(isoDate: string | null): Date | null { //..

const dateTypeExpected = mapIsoDate('2023-09-01');
//    ^? const dateTypeExpected: Date

const nullTypeExpected = mapIsoDate(null);
//    ^? const nullTypeExpected: null
```

* function overloading is **not** possible using arrow functions and **only** possible using function declarations

Function overloading

is further described in the TypeScript Handbook:

typescriptlang.org/docs/handbook/2/functions.html#function-overloads

(More) Accurate types

could be achieved using:

- declaration merging ✔
- generics and / or `keyof` keyword ✔
- function overloading ✔

➢ “type programming” 🤓

Type Programming in TypeScript

is (a bit) like programming… but then in the ‘type space’

- is facilitated by type parameters (generics)
- allows doing custom type inference
- by “creating types from types”

Type Programming in TypeScript

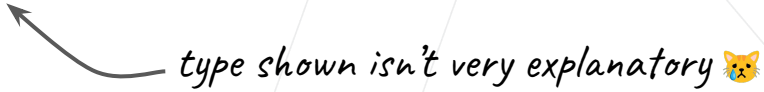
is easy using utility types like `Partial` and `Required`

```ts
type _1 = Partial<{ a: number; b: string }>;
//    ^? type _1 = { a?: number | undefined;
//                   b?: string | undefined; }

type _2 = Required<{ a?: number; b?: string }>;
//    ^? type _2 = { a: number; b: string; }
```

Type Programming in TypeScript

is powerful when combining utility types

```ts
type _ = Required<Pick<User, 'id'>> & Omit<User, 'id'>;
//    ^? type _ = Required<Pick<User, "id">> & Omit<Use…
```




```ts
interface User {
  id?: number;
  firstname: string;
  lastname: string;
}
```

Type Programming in TypeScript

is ever more easy using **third-party** utility types

```ts
type _ = Required<Pick<User, 'id'>> & Omit<User, 'id'>;
//   ^? type _ = Required<Pick<User, "id">> & Omit<Use…

interface User {
  id?: number; firstname: string;  lastname: string;
}
```

Type Programming in TypeScript

is ever more easy using **third-party** utility types

```ts
import { Simplify } from 'type-fest';

type _ = Simplify<Required<Pick<User, 'id'>> & Omit<User, 'id'>>;
//    ^? type _ = { id: number; firstname: string;
//                  lastname: string; }

interface User {
  id?: number; firstname: string;  lastname: string;
}
```

Type Programming in TypeScript

is ever more easy using **third-party** utility types

```ts
import { SetRequired } from 'type-fest';

type _ = SetRequired<User, 'id'>;
//   ^? type _ = { id: number; firstname: string;
//                 lastname: string; }

interface User {
  id?: number; firstname: string;  lastname: string;
}
```

Type Programming in TypeScript

is done by “Creating Types from Types” in various ways *

- generics and / or `keyof` keyword ✔
- using pre-written utility types ✔
- `typeof` keyword (from “type space”)
- indexed access types
- conditional types
- mapped types
- template literal types

* see also chapter of the TypeScript Handbook: https://www.typescriptlang.org/docs/handbook/2/types-from-types.html

`typeof` keyword (from **type space**)

retrieves type of variable or property

`typeof` keyword (from **type space**)

is often useful inside type argument of utility types

```ts
function createPoint(x: number, y: number) {
  return { x, y };
}

type Point = ReturnType<createPoint>;
// TS2749: 'createPoint' refers to a value, but is being used as a type here.
//         Did you mean 'typeof createPoint'?
```

`typeof` keyword (from **type space**)

is often useful inside type argument of utility types

```ts
function createPoint(x: number, y: number) {
  return { x, y };
}

type Point = ReturnType<typeof createPoint>;
//   ^? type Point = { x: number; y: number; }
```

Type Programming in TypeScript

is done by “Creating Types from Types” in various ways

- generics ✔
- `keyof` keyword ✔
- `typeof` keyword (from “type space”) ✔

➢ indexed access types

➢ conditional types

➢ mapped types

➢ template literal types

Live-demo with

Visual Studio Code

(and more)

(More) Accurate types

could be achieved using:

- declaration merging ✔
- generics and / or `keyof` keyword ✔
- function overloading ✔
- “type programming” 🤓 ✔

hopefully it wasn’t too overwhelming 😬

Nested Property getter

using conditional-, mapped- and template literal types 🤓

Live-coded with

Visual Studio Code

(and more)

* live-coding source-code can be found in `6-path.ts` and `7-path-value.ts` files in GitHub repository

Coding conventions

Naming conventions for type parameters:

- Single capital letter (old skool generics)
  - e.g. `T` and `R`
- Like regular types (modern TS Handbook)
  - e.g. `Type` and `Result`
- Like regular types, ***but*** with `T` prefix *
  - e.g. `TType` and `TResult`

* see video of Matt Pocock about this naming convention: https://www.totaltypescript.com/tips/how-to-name-your-types

Coding conventions

Naming conventions for utility types:

- name of utility type either:
  - is same as function name (e.g. `Trim`)
  - describes the end state (e.g. `Writable`)
  - describes its modification (e.g. `SetRequired`)
- predicate like types often use `Is` prefix
  - e.g. `IsAny`, `IsUnknown` or `IsNever`

Type Programming

is tempting to go overboard with:

- sometimes doesn’t make a lot of sense
  - like our `sort`/`getNestedProperty` 😜
  - since argument probably is no literal type
- types do nothing at runtime (type erasure)
- types are hard to debug and maintain

Type Programming

can still be taken advantage of:

- using `ts-reset` library
- using utility types (built-in and third-party)
- applying **simple** to understand type programming
  - e.g. basic conditional types
  - use `expect-type` to unit test your types

Type Programming

takes some time to really learn:

- read “Creating Types from Types” chapter:
  typescriptlang.org/docs/handbook/2/types-from-types.html
- look at open-source projects (e.g. type-fest)
- practice doing Type Challenges
  github.com/type-challenges/type-challenges

TypeScript Inaccuracies

probably will never be fixed for the full 💯%:

- but hopefully you learned some ways to improve

TypeScript Inaccuracies

scan QR code for slide-deck, source-code and link to training info

C'est Fini.

divotion **training:**

TypeScript Essentials - Yin-Yang ☯

- 16 january
- 7 februari
- 21 maart