TypeScript type-guards 

#whoami

Charly POLY - Senior Software Engineer at 

#whoami

Charly POLY - Senior Software Engineer at algolia

- writing TypeScript Essentials series on

There will be 9 chapters :

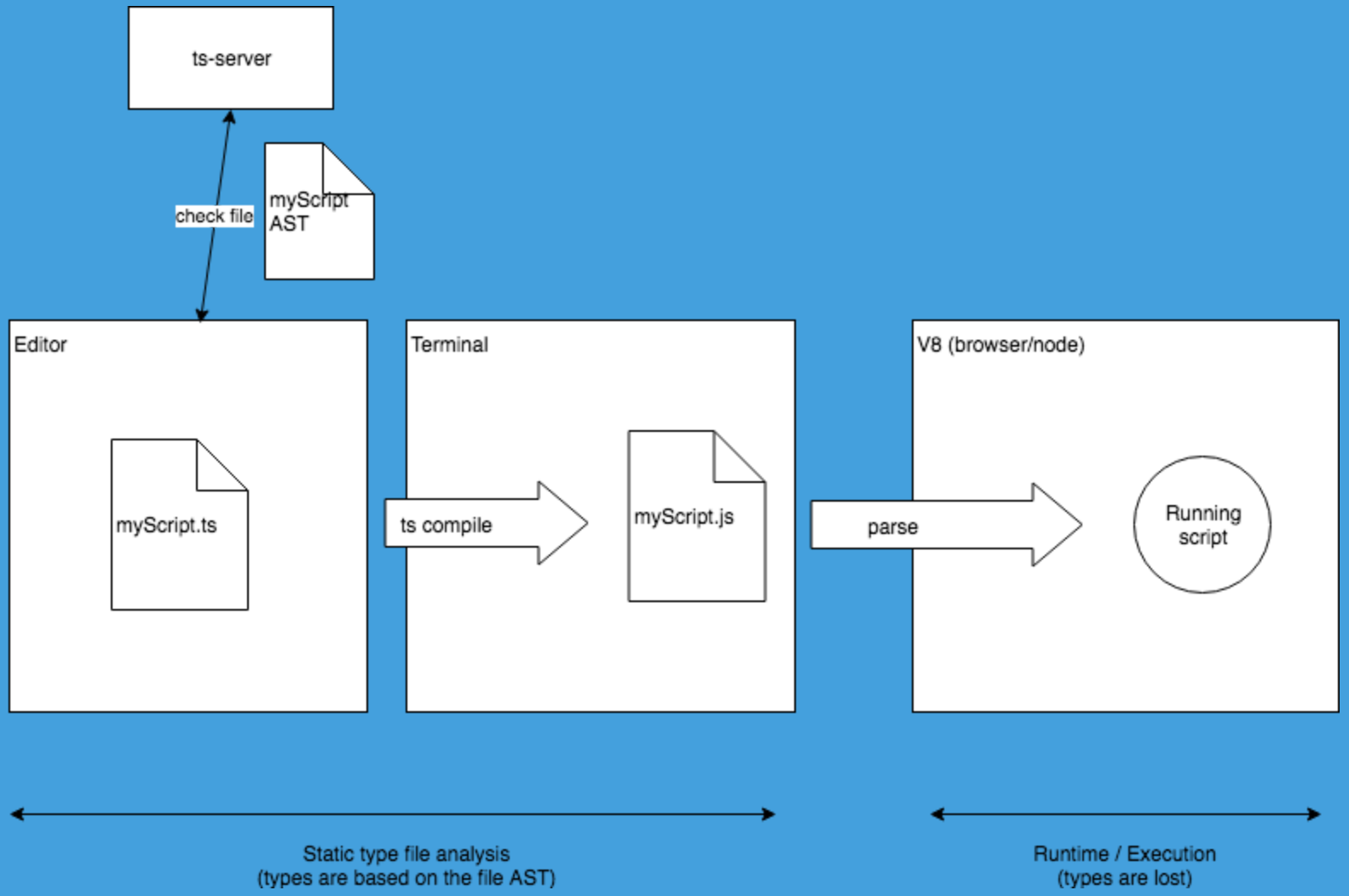
- Why use TypeScript, good and bad reasons ⚖️
- The honest trailer: pros and cons 📺
- Learn the basics 📖
- Generics and overload 🤦‍♂️
- Super-types 💪
- Make types “real”, the type guard functions 💂‍♂️
- Tooling: webpack, TSlint ⚙️
- When and how to start using TypeScript in production
- Writing a NPM module (2019 version) 📦

Plan

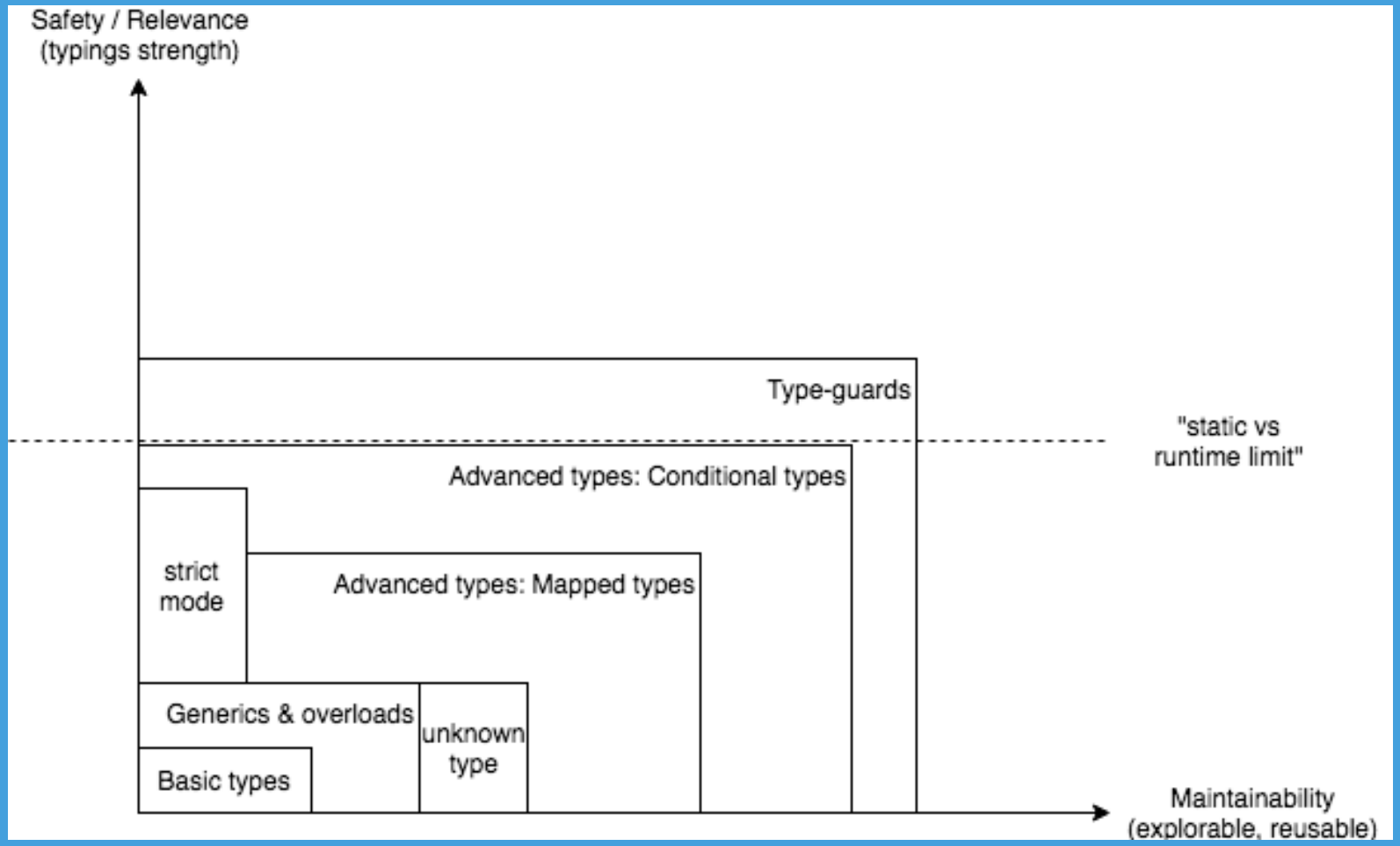
- "Types are not real": static vs runtime
- TypeScript embed type-guards
- The Discriminated Unions
- User-Defined type-guards
- How to start?

"Types aren't real"

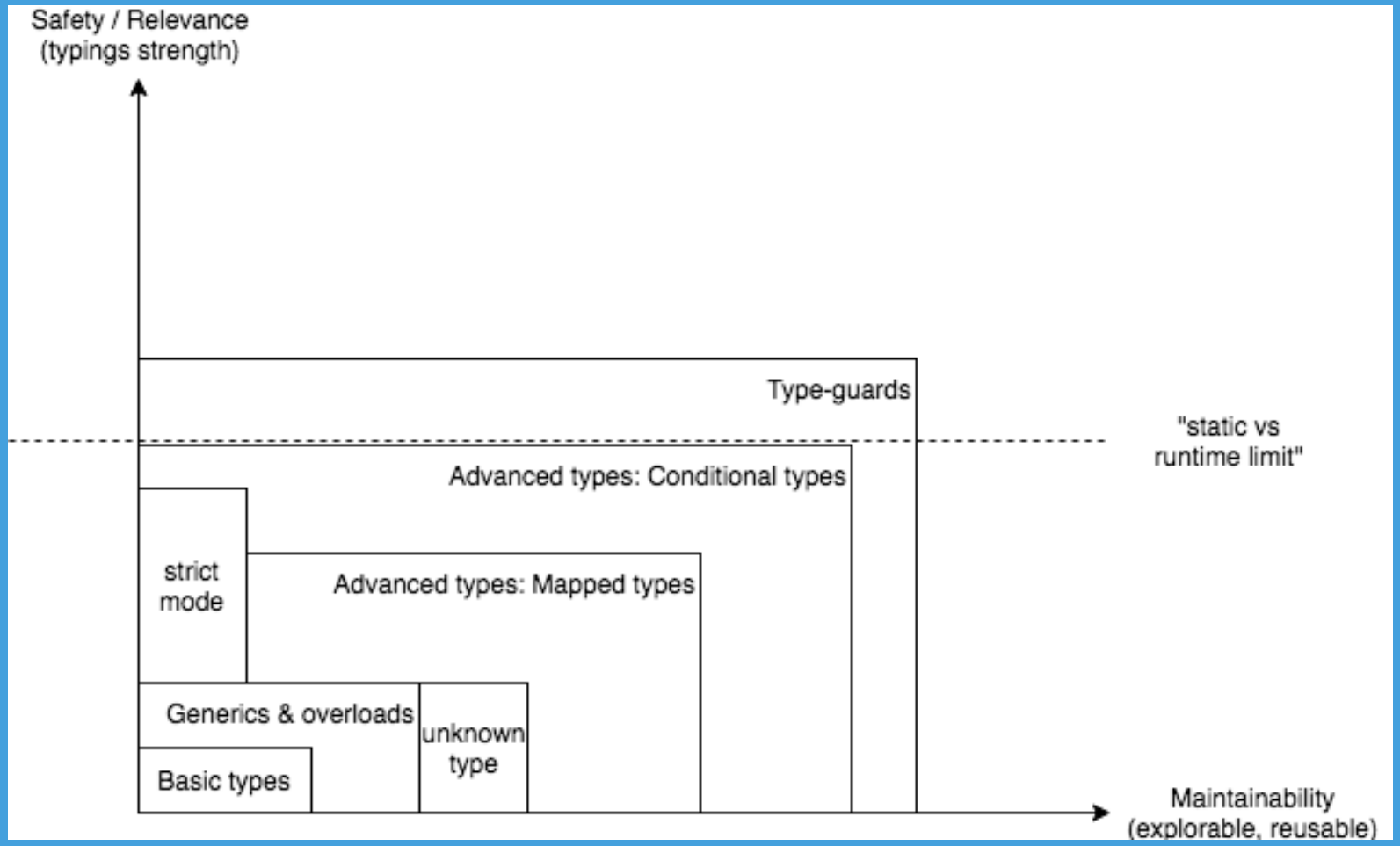
"Types aren't real"



Make "types" stronger



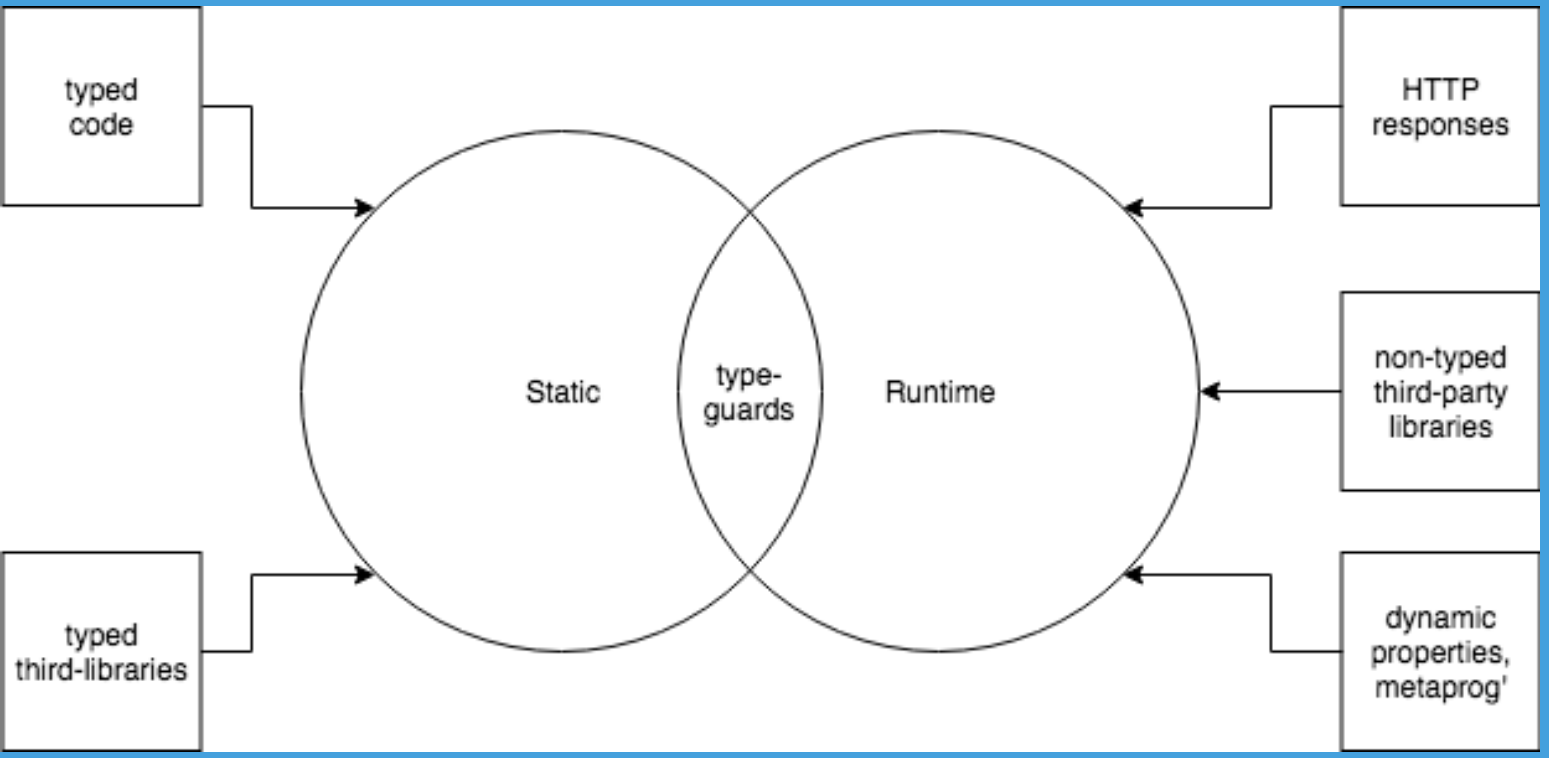
Make "types" stronger



"static vs runtime limit":

- any to a "defined type"
- "as" keyword

Typing "weak spots"




```
const chat: Chat = await http.post(
  `/chats`,
  { id }
).then(
  response => response.body.chat as Chat
);


// ...

export const Form = reduxForm<any, any>(
  /* ... */
)
```

"Make types reals"

> *A type guard is some expression that performs a **runtime check** that guarantees the type in some scope.*

TypeScript Handbook, "Advanced types"

TypeScript embed type-guards

typeof type-guards

```
function formatMoney(amount: string | number): string {
   let value = amount; // value type is number or string
   if (typeof amount === "string") {
      value = parseInt(amount, 10); // amount type is string
   }
   return value + " $"; // value type is number
}

// calling formatMoney({ myObject: 1} as any)
// will not call parseInt with an object
```

> *TypeScript and JavaScript runtime are now tied to the same behaviour.*

TypeScript embed type-guards

Embed type-guards:

- **typeof** operator
- **instanceof** operator
- **in** operator

The Discriminated Unions

The Discriminated Unions

> *Discriminated Unions is a pattern that allow to build types that shares a common property but have different shapes*

The Discriminated Unions

> *“ Discriminated Unions is a pattern that allow to build types that shares a common property but have different shapes*

Example with redux actions

```
interface Action {
  type: string; // the discriminant
}

interface ActionA extends Action {
  type: 'ActionA';
  mypropA: string;
}

interface ActionB extends Action {
  type: 'ActionB';
  mypropB: string;
}

type anyAction = ActionA | ActionB; // the union
```

1. Types that have a common, singleton type property — **the discriminant**.
2. Then, a type alias that takes the union of those types — **the union**.
3. Finally, a type guard on the common property (on the discriminant).

TypeScript Handbook, "Advanced types"

The Discriminated Unions

```
interface Action { type: string; }

interface ActionA extends Action {
  type: 'ActionA';
  mypropA: string;
}

interface ActionB extends Action {
  type: 'ActionB';
  mypropB: string;
}

type anyAction = ActionA | ActionB;

// ...

function reducer(state: State, action: anyAction): Action {
  switch(action.type) { // "ActionA" | "ActionB"
    case 'ActionA':
      return { ...state, prop: action.mypropB }; // TS ERROR!
      break;
    case 'ActionB':
      return { ...state, prop: action.mypropB }; // OK
      break;
    default:
      return state;
  }
}
```

The Discriminated Unions

```ts
interface Person { name: string; }

interface Car {
    type: 'car';
    passengers: [Person] | [Person, Person] |
    [Person, Person, Person] | [Person, Person, Person, Person]
}

interface Moto {
    type: 'moto';
    passengers: [Person] | [Person, Person]
}

type Vehicule = Car | Moto;

let v: Vehicule | undefined;

switch (v.type) {
    case 'moto':
        v.
        br passengers
           type
    default:
        break;
}
```

(property) Moto.passengers: [Person] | [Person, Person]

User-defined type-guards

> *"real world usage" of TypeScript is not restricted to scalar types (**string**, **boolean**, **number**, etc...).*
> *Real world applications mainly deals with complex **object** or **custom types**.*
>
> *This is when "**User-Defined Type Guards**" help us.*

TypeScript — Make types "real", the type guards

User-defined type-guards

```
function isFish(pet: any): pet is Fish {
    return pet.swim !== undefined;
}
```

- guard function **argument type**, like for overloads, should be **as open as possible**.

- a new **is** operator, called type predicate.

User-defined type-guards

```ts
interface Fish {
    swim: () => [number, number, number]
}

interface Cat {
    walk: () => [number, number]
}

function isFish(pet: Fish | Cat): pet is Fish {
    return (<Fish>pet).swim !== undefined;
}

let a: Fish = {} as any;

if (isFish(a)) {
    a.
}
```

```
swim  (property) Fish.swim: () => [number, numb…
```

User-defined type-guards

Good points of User-Defined type-guards:

- matches real-world expectations
  - more flexible
  - support complex types
- stateless and isolated ➡️ testable

Where to start?

Avoid "any" and "as" and use type-guards for complex use-cases

Where to start?

Avoid "any" and "as" and use type-guards for complex use-cases

```
if (!!myobject.someProp) {
  (<MyType>myobject).someMethod()
}
```

➡️ User-Defined type-guards

Where to start?

Avoid "any" and "as" and use type-guards for complex use-cases

```
if (!!myobject.someProp) {
  (<MyType>myobject).someMethod()
}
```

➡️ User-Defined type-guards

```
function reducer(state: State, action: any): Action {
  switch(action.type) { // "ActionA" | "ActionB"
    case 'ActionA':
      return { ...state, prop: <ActionA>action.mypropA };
      break;
    // ...
    default:
      return state;
  }
}
```

➡️ Discriminated Unions

Where to start?

Save time while building type-guards by using io-ts

Introduced in "Typescript and validations at runtime boundaries" article by @lorefnon, **io-ts** is an active library that aim to solve the same problem:

> *TypeScript compatible runtime type system for IO decoding/encoding*

Where to start?

io-ts overview

```
const Person = t.interface({
  name: t.string,
  age: t.string
})


interface IPerson extends t.TypeOf<typeof Person> {}

// same as
//   interface IPerson {
//     name: string
//     age: number
//   }


let a: any = {};

if (Person.is(a)) {
  // a is a Person type
}
```

Powerful API:

- decode()
- encode()
- is()

and also,

- custom error reporters
- unions and recursives types support

Conclusion

- Types can be real
- Avoid "as" operator, use type-guards
- TypeScript is more powerful than you think

Thanks for listening!

For more, look at the "TypeScript — Make types “real”, the type guards" article

🔗 honest.engineering

 @whereischarly

 /wittydeveloper