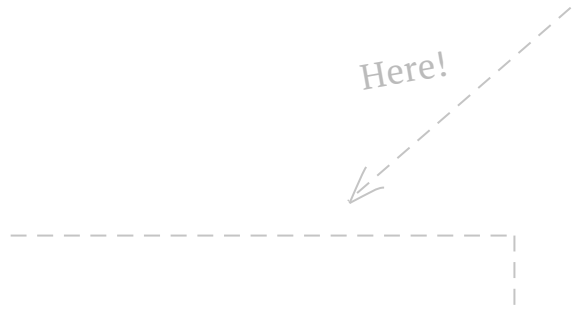
Navigation

Hover on the bottom-left corner to see the navigation's controls panel, learn more

Keyboard Shortcuts

`right` / `space`	next animation or slide
`left` / `shift` `space`	previous animation or slide
`up`	previous slide
`down`	next slide



Intro to io-ts

or: *How my trust issues manifest in code*

Press Space for next page →

Before we begin

- 'dudes' and 'guys' have been my preferred collective pronouns (ie when addressing groups) for decades, and I'm probably going to use them during this talk.
  - I don't intend any harm; my apologies if this habit will be uncomfortable for you.
- Who am I?
  - Arthur
  - Just some dude on the internet.
  - I solve other peoples problems, professionally.

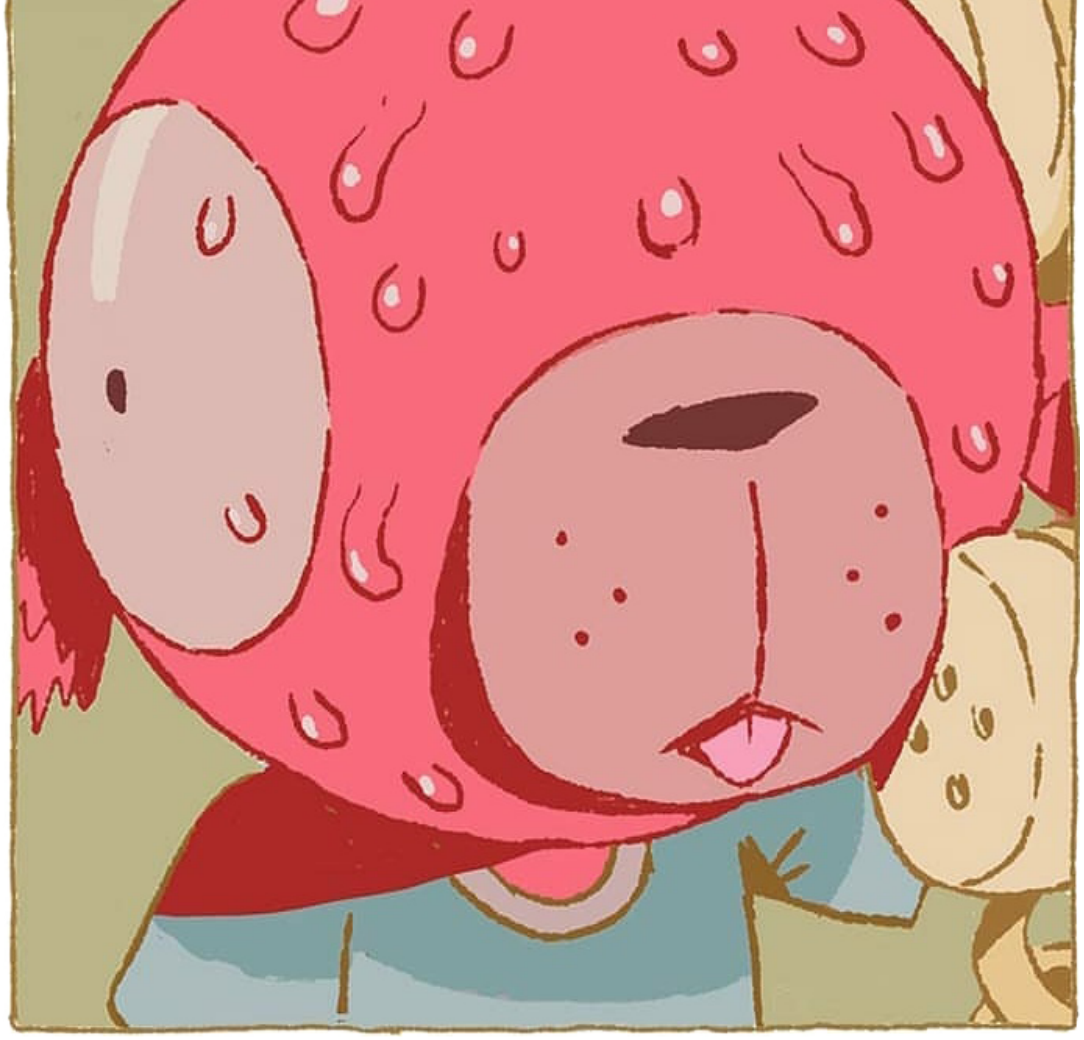
what we're solving

- type safety
- runtime representation of static types
- shotgun parsing

What we're solving: Type safety

```
// it's probably fine
const foo = JSON.parse(scaryInput) as Optimism;
```

- successful `JSON.parse` results in a valid js object, but we don't know anything else about that object
  - and thus its output type is `any`
- `any` is a highly infectious disease.
- casting an `any` isn't solving the problem, it's silencing the compiler.



What we're solving: Runtime representation

```
type Optimism = {
  // …
};
```

- type systems are fantastic at compile time…
- …but eventually our software has to run, and we need our data to be correct at *runtime*
- we *should* be able to say that anything within our domain is valid
- to reach that point, we need to parse at the domain boundary

What we're solving: Shotgun parsing

> Shotgun parsing[1] is a programming antipattern whereby parsing and input-validating code is mixed with and spread across processing code—throwing a cloud of checks at the input, and hoping, without any systematic justification, that one or another would catch all the "bad" cases.

Which

> …deprives the program of the ability to reject invalid input instead of processing it. Late-discovered errors in an input stream will result in some portion of invalid input having been processed, with the consequence that program state is difficult to accurately predict.

1. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them

Show me

```
import * as t from 'io-ts';

// define a codec to use at runtime
const User = t.type({
  id: t.number,
  name: t.string,
})

// generate a type from that codec
type User = t.TypeOf<typeof User>

// attempt to decode an input
const obj = JSON.parse(scaryInput) // any
const parsed = User.decode(obj)    // Either<Errors, User>

if (isLeft(parsed)) {
  // handle bad data. scaryInput was not a valid User.
  // ...
  return;
}

// we know it's a valid User AND so does the compiler!
const user = parsed.right         // User
```

End...?


AAH,
SO IT'S LIKE THAT, HUH. I UNDERSTAND EVERYTHING NOW.
DOESN'T GET IT AT ALL

Foundational knowledge

- knowledge of typescript is assumed
- io-ts is part of the fp-ts ecosystem
- because its built on a functional programming foundation, it’s useful for you to know at least a trivial amount about fp.
- bits we will cover
  - Either type
    - and its refinements
  - Validation type
  - Basic io-ts usage
- bits we won’t cover
  - typeclasses, higher kinded types, and so much more

Foundational knowledge:

Either type

Foundational knowledge: Either type

- a common algebraic data type (ADT). It does cool stuff.
- to start with, think of it as a container
- disjoint union; it is *Either* a `Left` or a `Right`

```
type Either<E, T> =
    | Left<E>
    | Right<T>
```

  - the ts implementation is specifically a discriminated union

by convention:

Left

`Left` is used for error or failure states.

Contains a value in its `.left` prop

Right

`Right` is used for valid or success states.

Otherwise known as the happy path.

Contains a value in its `.right` prop

Foundational knowledge: Either type

Similar patterns in other modern languages

```rust
// Rust has a wonderful type system that includes Result. It uses Ok and Err
// instead of Left and Right, but it's effectively the same thing.
fn foo() -> Result<String, io::Error> { ... }

let bar = foo();
// non-idiomatic, but relevant for this audience
if (bar.is_err()) {
  // ...
}
```

```go
// Go uses multiple return values. Not a proper Either, but the usage pattern
// has some similarity. It's more of a Pair/Tuple than an Either.
func foo() (string, error) { ... }

msg, err := foo()
if err != nil {
  // ...
}
```

Foundational knowledge: Either type

- Either is type, not a class. It maintains a clean separation between data and behavior.
- fp-ts provides a collection of functions that operate on Either
- we'll only focus on a few of them, namely the `` `isLeft` `` and `` `isRight` `` refinements

Foundational knowledge: Either type refinements

isLeft

> Returns `true` if the Either is an instance of `Left`, `false` otherwise.

signature

```
export declare const isLeft: <E>(ma: Either<E, unknown>) => ma is Left<E>
```

- Using this type refinement lets the compiler (and readers) know that
  - this specific instance of `Either` is a `Left`
  - and has a `.left` property.

```
decoded.left; // Error, Property 'left' does not exist on type Either

if (isLeft(decoded)) {
  handleError(decoded.left); // Ok. Decoded is a Left, so it has a left prop
  return;
}
```

Foundational knowledge: Either type refinements

isRight

> Returns `true` if the Either is an instance of `Right`, `false` otherwise.

signature

```
export declare const isRight: <A>(ma: Either<unknown, A>) => ma is Right<A>`
```

- Using this type refinement lets the compiler (and readers) know that
  - this specific instance of `Either` is a `Right`
  - and has a `.right` property.

```
decoded.right; // Error, Property 'right' does not exist on type Either

if (isRight(decoded)) {
  doStuff(decoded.right); // Ok. Decoded is a Right, so it has a right prop
}
```

Foundational knowledge: Validation type

Foundational knowledge: Validation type

name brand Eithers

`Validation` is an Either with a specific error type

```
type Validation<A> = Either<ValidationError[], A>
```

io-ts codecs will return a Validation

```
const validation = User.decode(possibleUserData)

if (isLeft(validation)) {
  const errList = validation.left  // ValidationError[]
  // ...
}

const user = validation.right      // User
```

Defining a codec

finally, parsing!

io-ts provides codecs for primitive types

```
const StringCodec = t.string
const NumberCodec = t.number
// etc...
```

and combinators that allow you to compose more complex codecs

```
// plain old js object.
// type User = { id: number; name: string; }
const User = t.type({
  id: t.number,
  name: t.string,
})

// type Foo = User & { phone: string }
const Foo = t.intersection(
  [
    User,
    t.type({
      phone: t.string
    }),
  ]
)
```

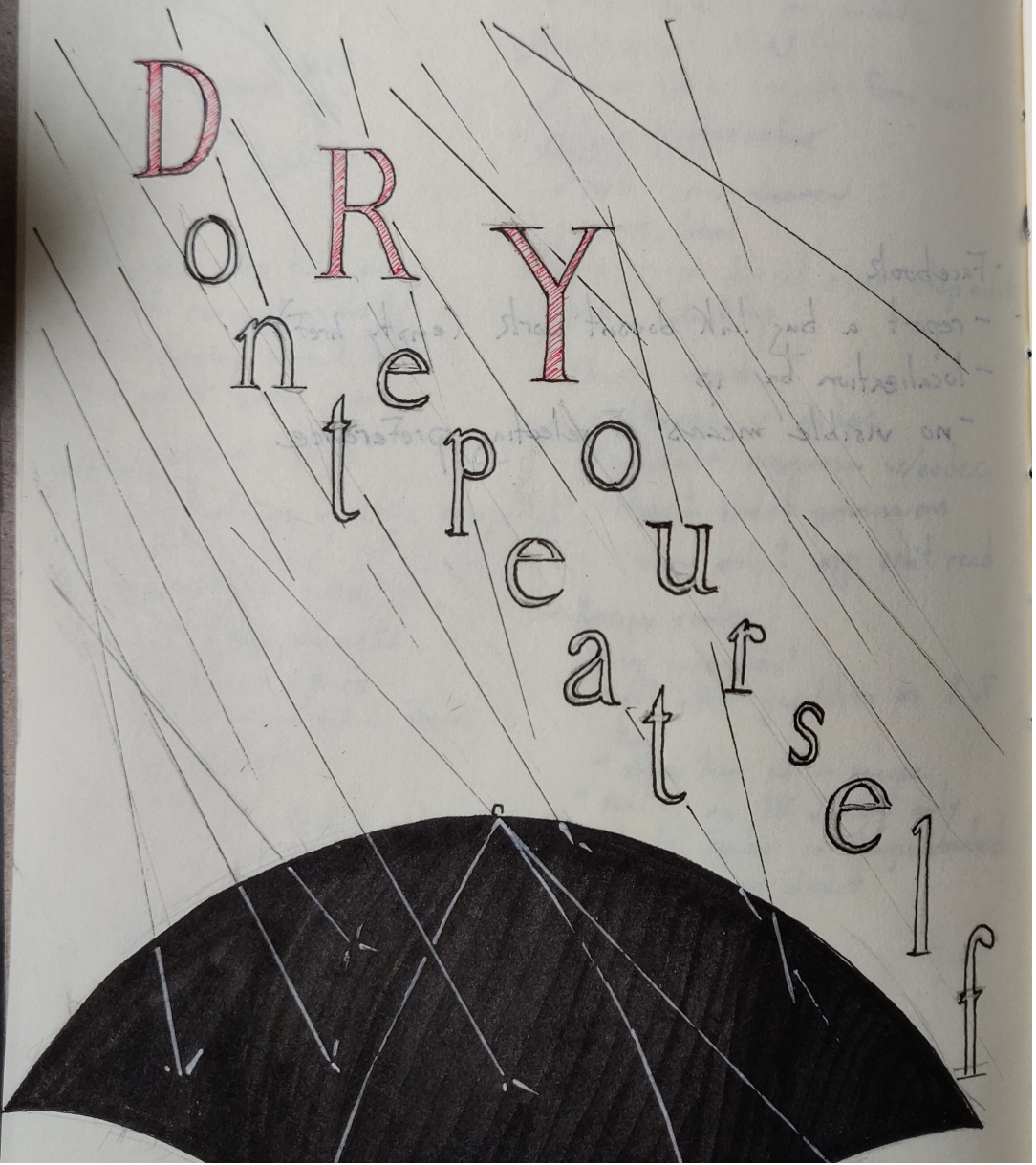
Defining a codec

but wait, there's more

We can also supply a name for our type.

This provides us with better human readable runtime errors when we leave the happy path.

```
const User = t.type(
  {
    id: t.number,
    name: t.string,
  },
  'User',
);
```



Extracting a type from a codec

stay DRY

Our codec definition also provides a static type (with a single additional line of boilerplate code).

```
const User = t.type({
  id: t.number,
  name: t.string,
});

type User = t.TypeOf<typeof User>
```

If our type or parsing needs to change, we only need to update one location.

There's no need to update the types in `types.ts`, *then* hunt down every instance of validation and update it as well.

Error reporting

if you see something, say something

We can handle `ValidationErrors` in infinitely many ways, but a reasonable starting point is to use the `PathReporter` included with io-ts.

```
import { PathReporter } from 'io-ts/lib/PathReporter'

const result = User.decode({ name: 'Giulio' })

console.log(PathReporter.report(result))
// => [ 'Invalid value undefined supplied to : { userId: number, name: string }/userId: number' ]
```

If we define a name in our codec this output becomes

```
// => [ 'Invalid value undefined supplied to : User/userId: number' ]
```

Not a huge difference with such a small type, but increasingly useful as type complexity increases.

It may be worth noting that `PathReporter` takes the entire `Validation`, not the `left` value.

Further reading

rtfm or gtfo

Directly applicable

- io-ts index docs
- fp-ts Either docs
- fp-ts Discord
  - It's a great place to improve your knowledge by solving other people's problems, or to just get help with your own question.

Supplementary

- Parse, don't validate
- Railway Oriented Programming
  - recorded talk
  - slides etc
- Mostly adequate guide to FP
- Scala cats docs
  - I'm not a mathlete; I find it much easier to view many fp concepts in terms of interfaces, and I've found the cats docs quite useful in that regard.

Questions

tell me how I've failed to teach you without saying I've failed to teach you