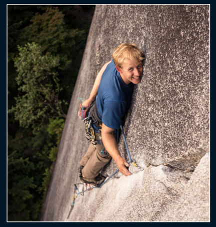
Effective Typed JavaScript



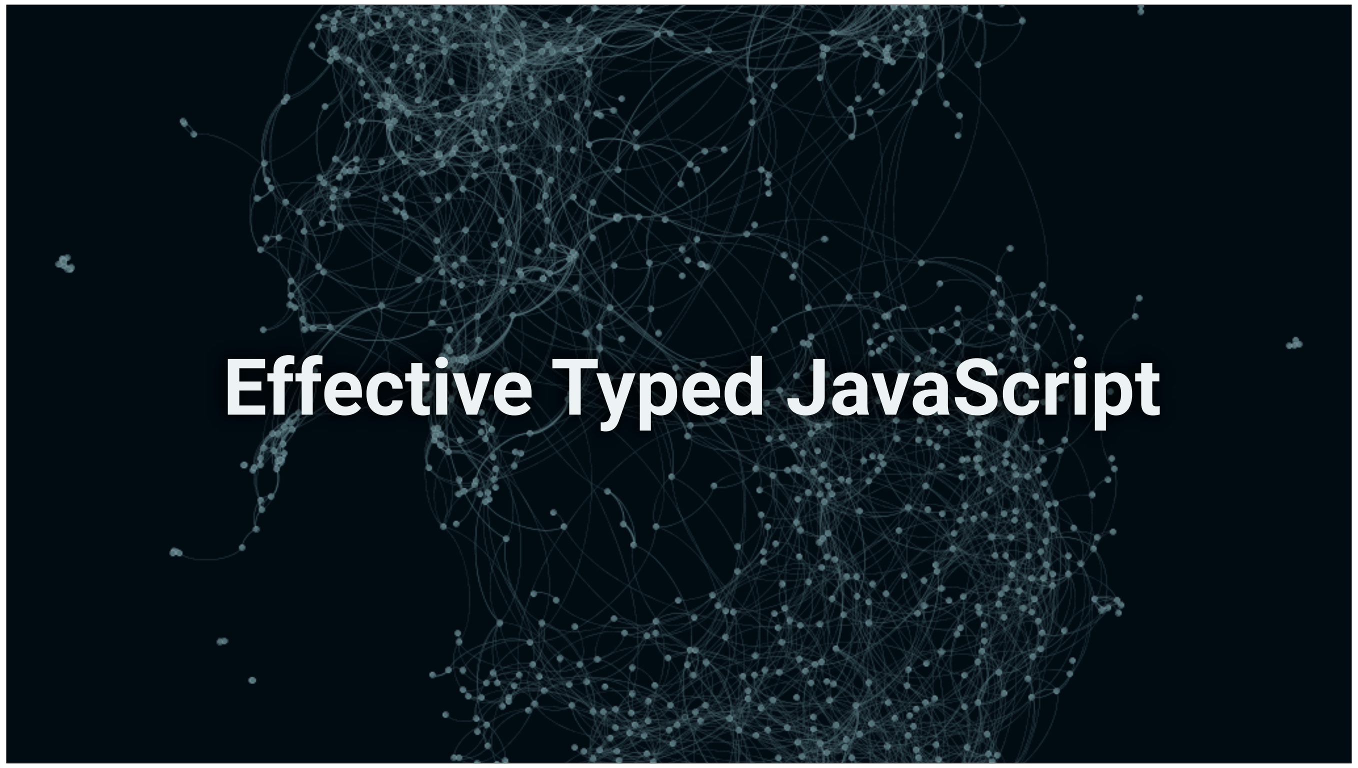
Languages Team

@jeremymorrell

Hi I'm Jeremy

I'm the Node owner at Heroku. That may sound cool,
That mainly involves writing bash and debugging other peoples' webpack configs

But sometimes I get to come out and talk to cool people like you



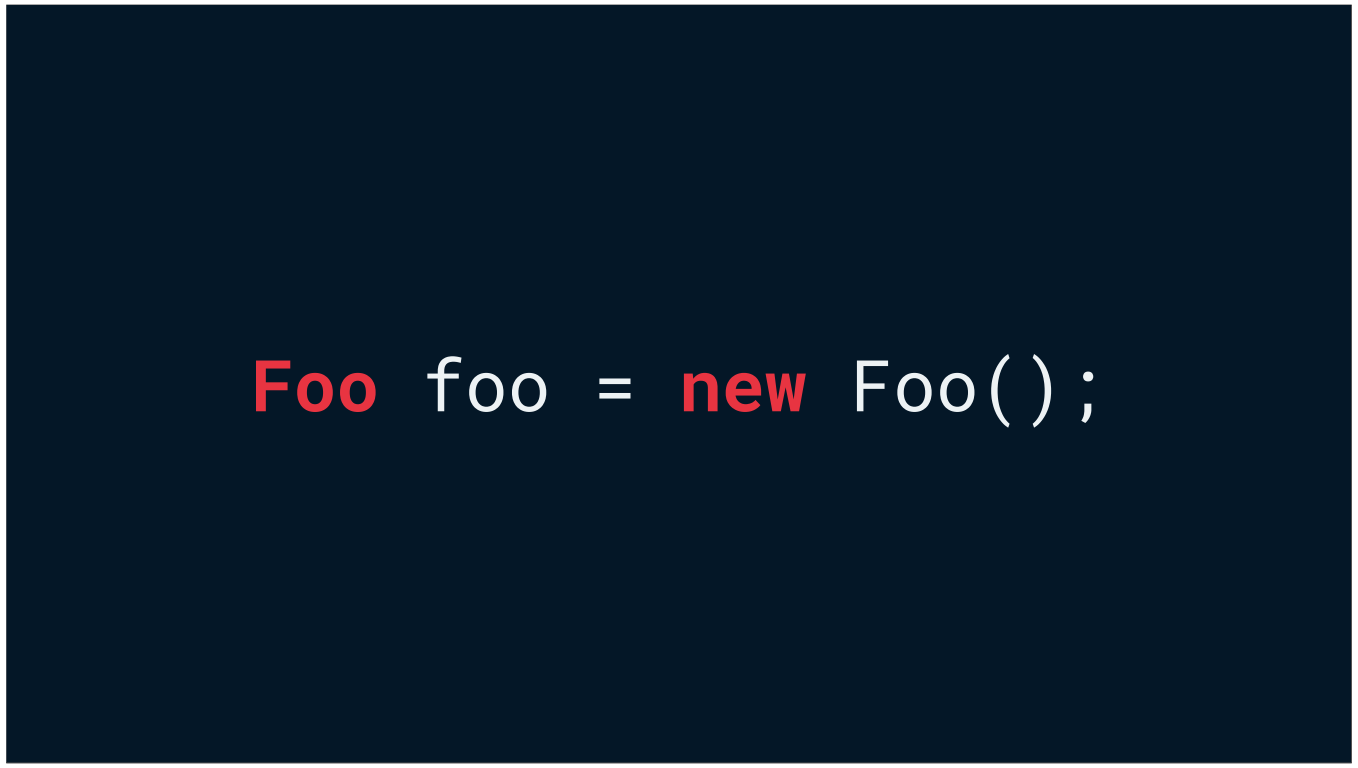
Effective Typed JavaScript

I've had types on my mind for a while

And the typescript talk made me want to talk about how my thinking on types has changed

It was a good talk over the mechanics
"this is a string", "this is a number"
but I wanted to dive into "thinking in types"

Some of you might be like I was a couple of years ago, when you hear the phrase Type System

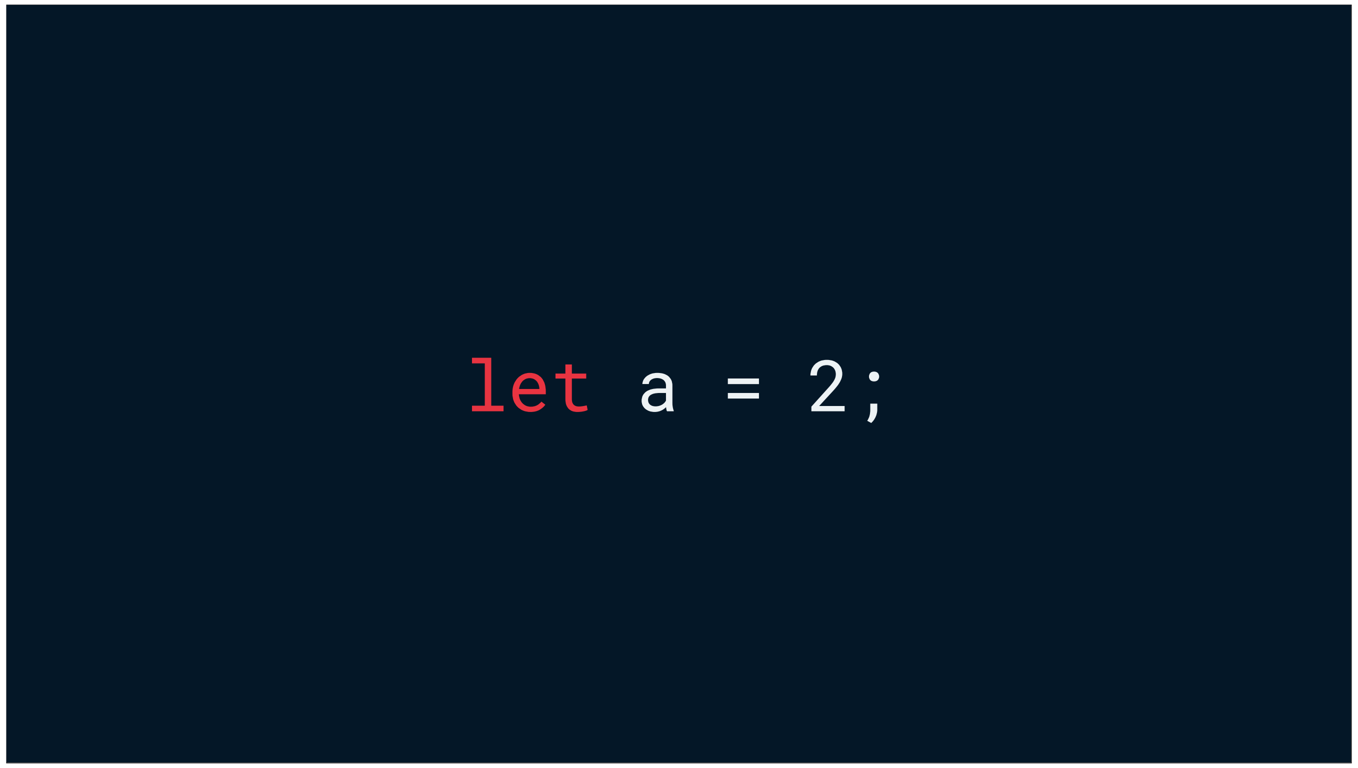


```
Foo foo = new Foo();
```

Ceremony

I never wrapped my head around the kinds of patterns you see in OOP langs
I never knew when to make a factory
Or a singleton
Or an abstractSingletonFactoryBean

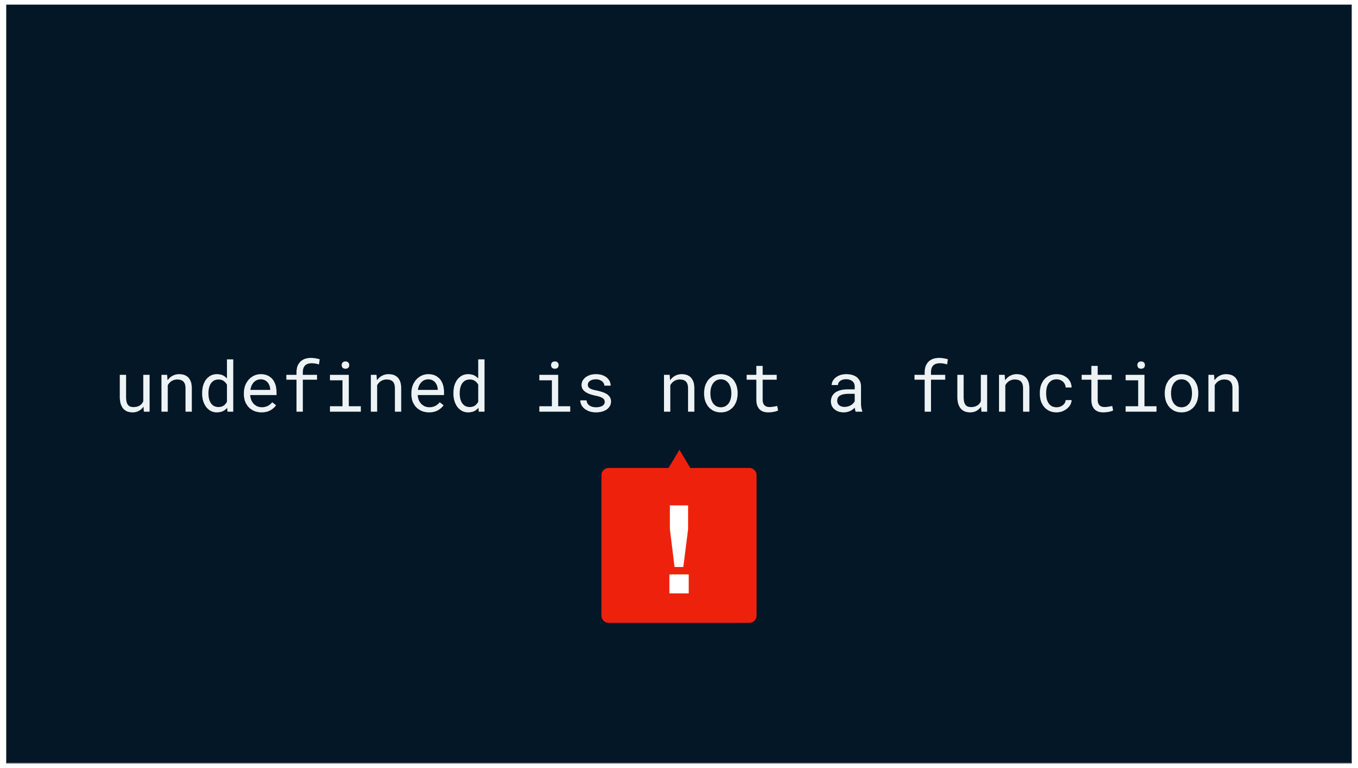
It never clicked in my brain



```
let a = 2;
```

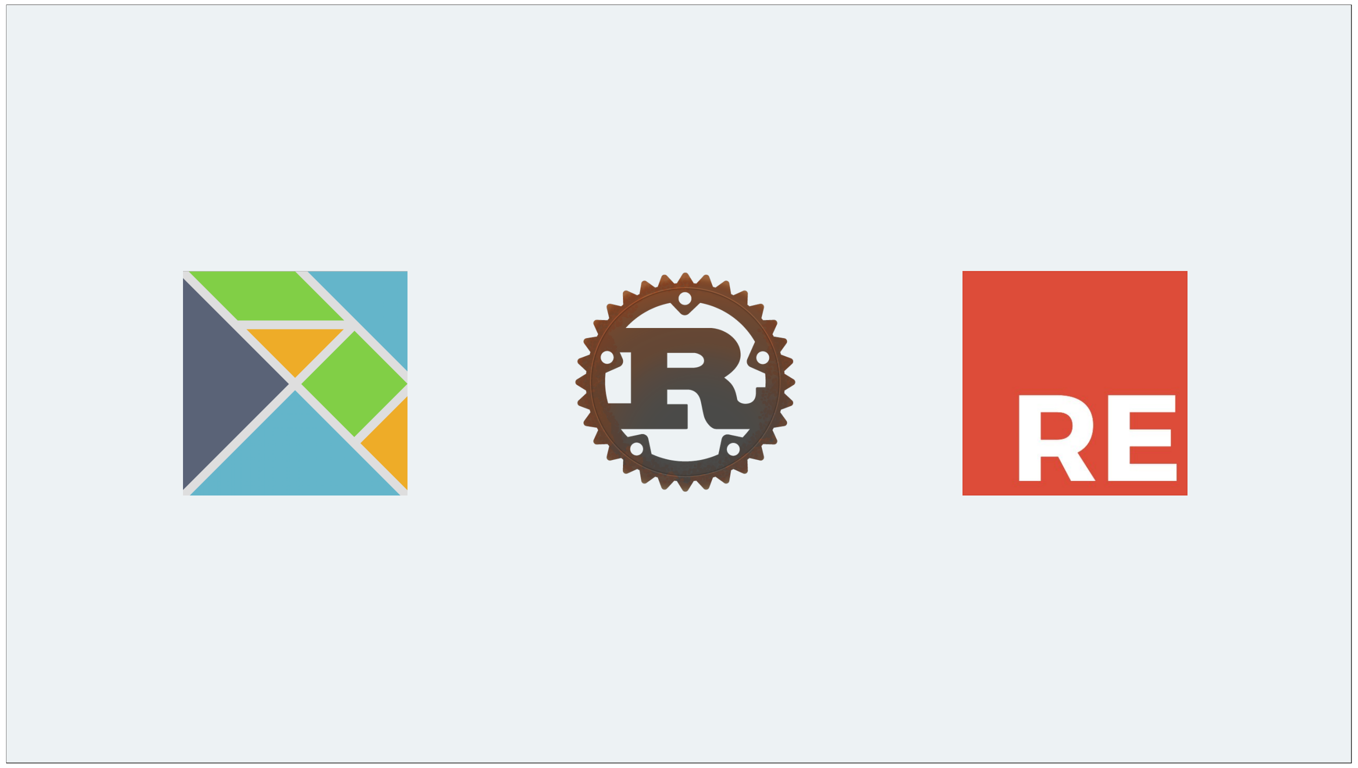
So when I found dynamic languages, it felt freeing

There was no ceremony.
You could just through things into objects or maps.
Everything was just objects and functions
Just focused on getting things done.



undefined is not a function

Sometimes you make mistakes, typos, or forgot to update something
when you added a new feature, and things blew up when the code ran,
`undefined is not a function` but that was your fault,
you forgot to handle a case
you mistyped a word
you didn't think through everything thoroghly enough
The computer is yelling at you
Thinking like a computer
fix it, maybe write a test, move on.
Worth it for productivity

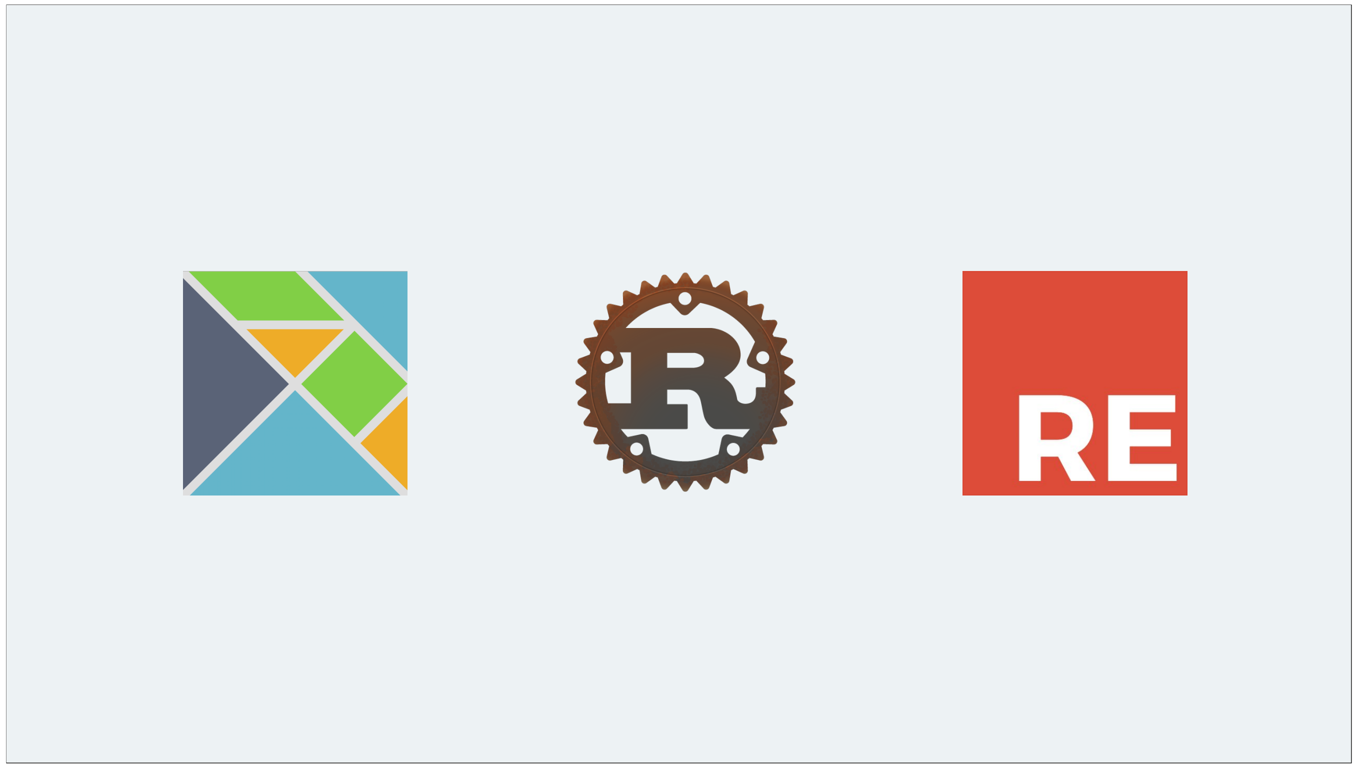


- In the past few years I've noticed an increasing interest in typed languages

Elm
  - No runtime errors. Well that sounds really nice.
  - Really excellent error messages
  - Focused on user experience

- Mozilla is rewriting Firefox piece by piece in Rust
  - which provides some interesting guarantees through its type system

- React community is starting to talk about this Reason thing
  - Isn't that really OCaml? Some academic language?
  - I write webservers. I don’t study PL theory



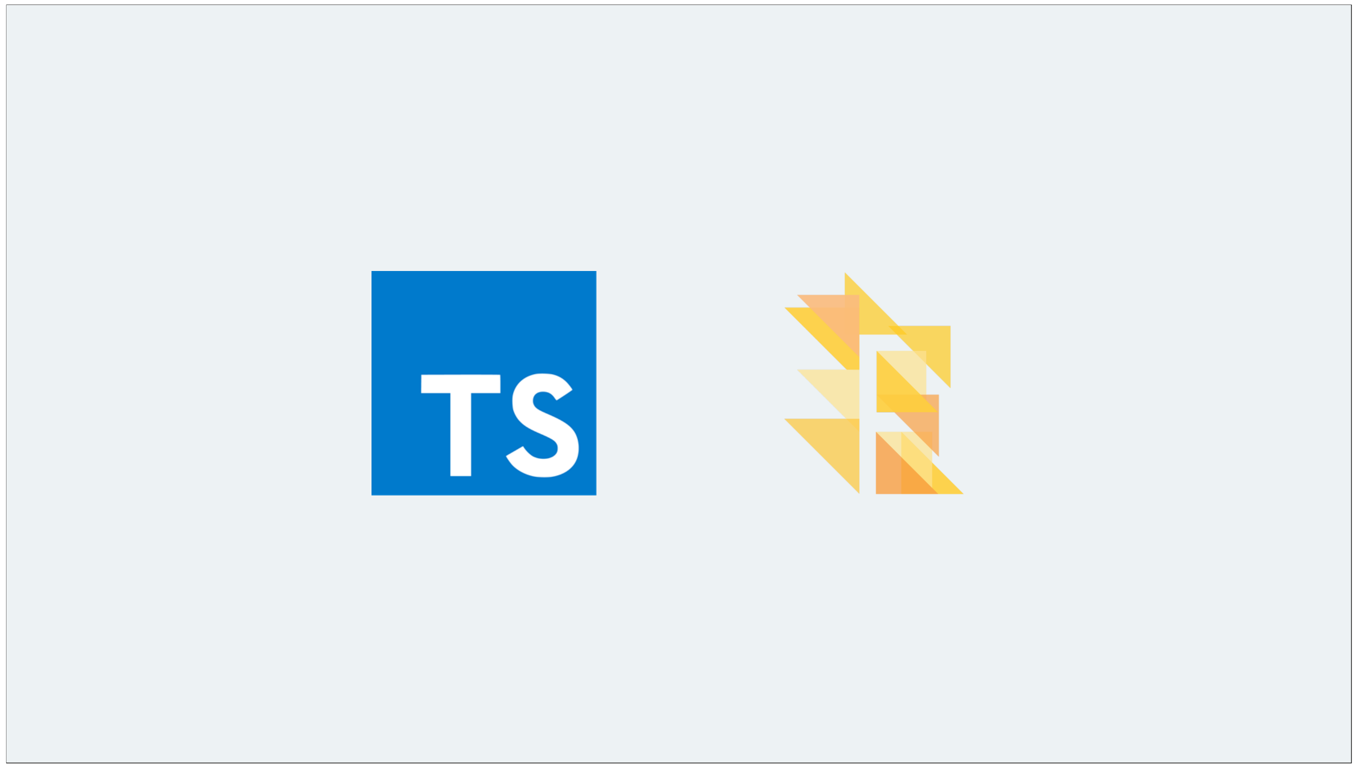
Ignoring these three, most new languages that are taking off have a type system of some sort:
- Go
- Swift
- Kotlin
- Elixir

Python and PHP are adding optional type systems

It seems like this is where our industry is headed
No new dynamic languages

Maybe there's something to this type system stuff after all.

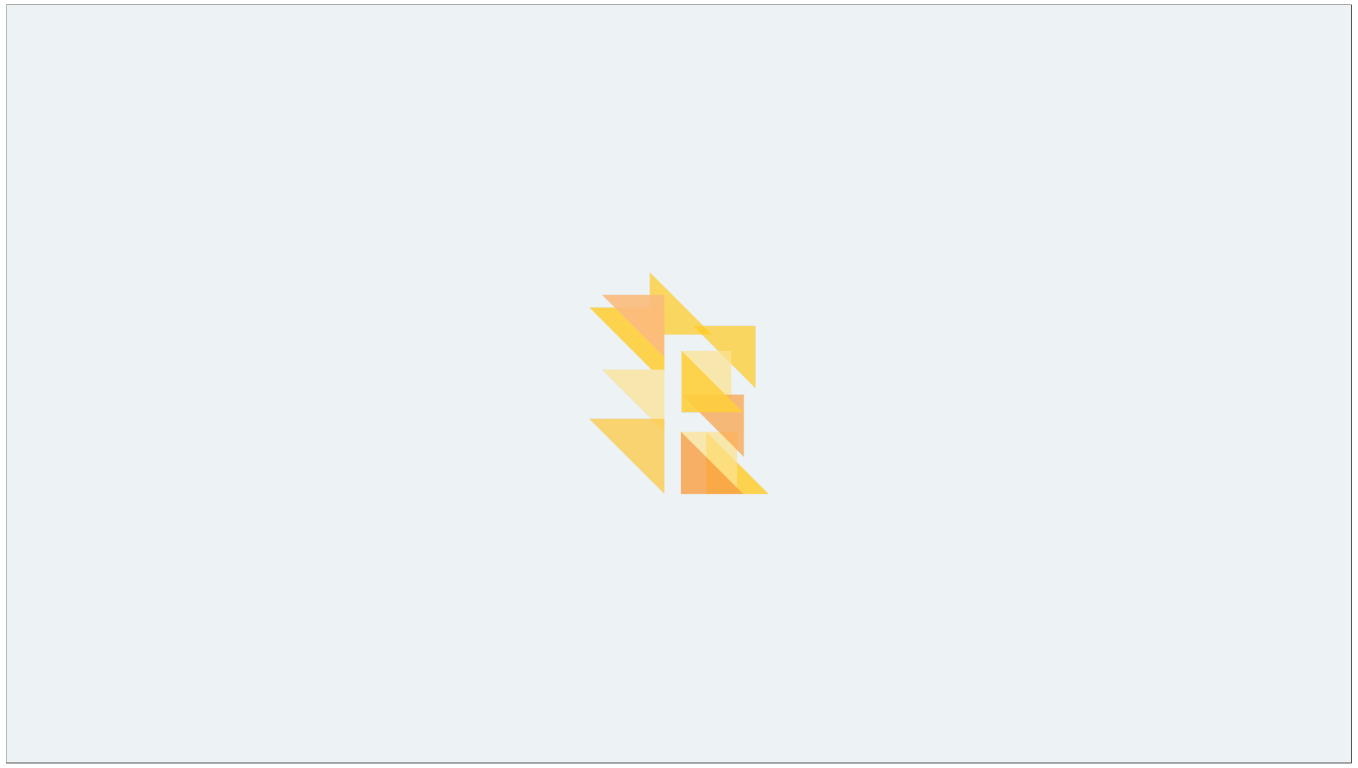
So all that is well and good, but I do javascript,
my team does javascript
our app is written in javascript,
thousands and thousands of lines of javascript



In that case, you're not out of luck
Flow and TypeScript are two projects that lay a type system on top of the language
we all know and love
Or know and begrudgingly accept

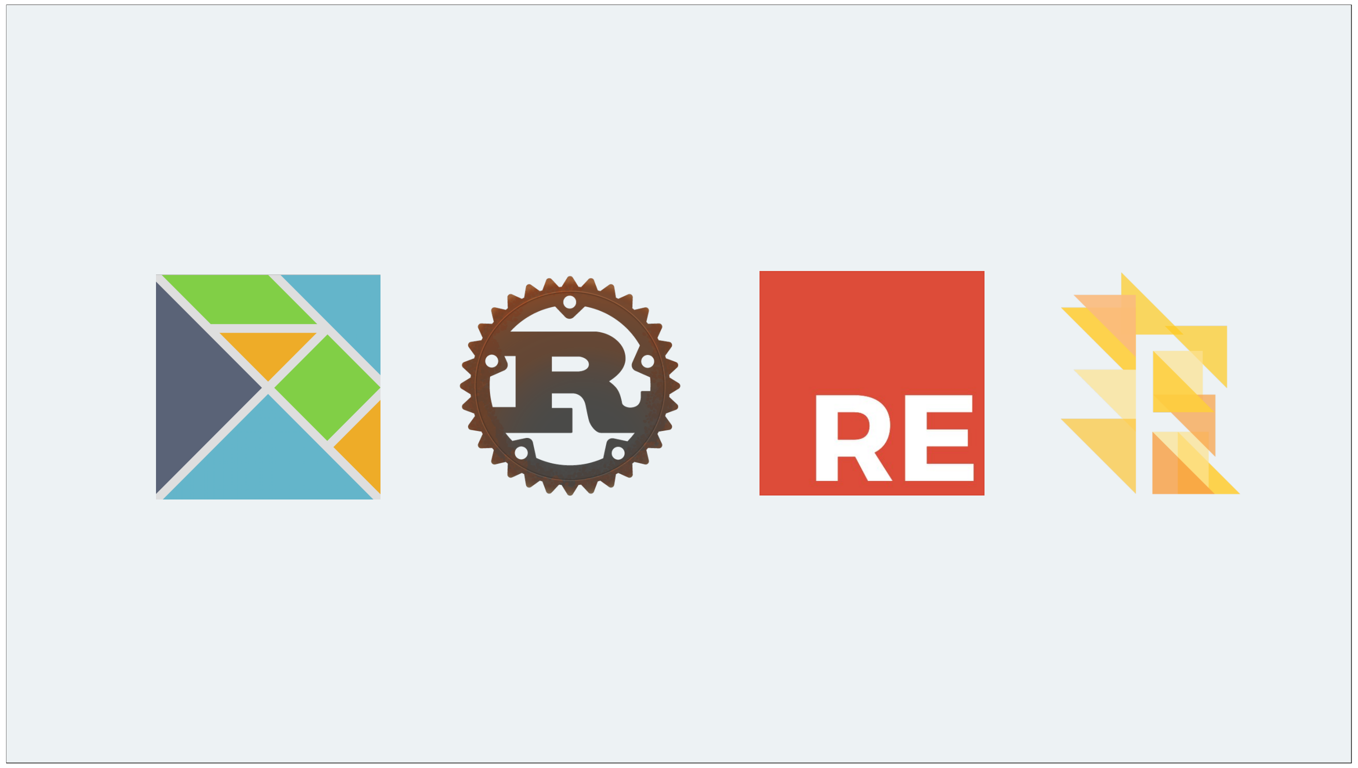
And if you're sitting there thinking
We just spent a week getting our webpack config just right
No way we're adding another tool
I hope this talk will help you see the value it might bring

Neither is perfect. I suggest trying both.



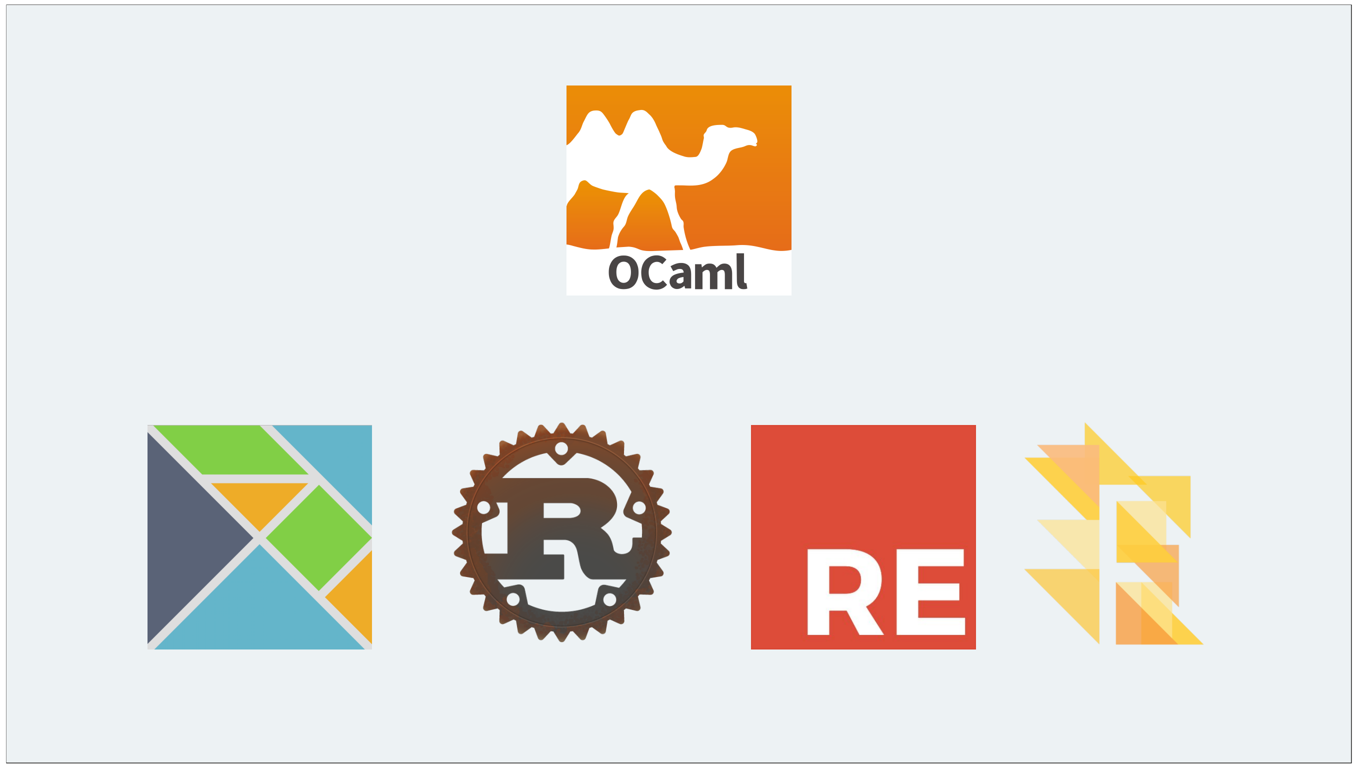
This is a short talk, so today I'm only going to talk about Flow
And use flow’s syntax

but everything here
Also applies to TypeScript
With some minor syntax differences
if you enable strict null checks and > 2.0



So this a few months ago this was my thinking

These languages all have something in common

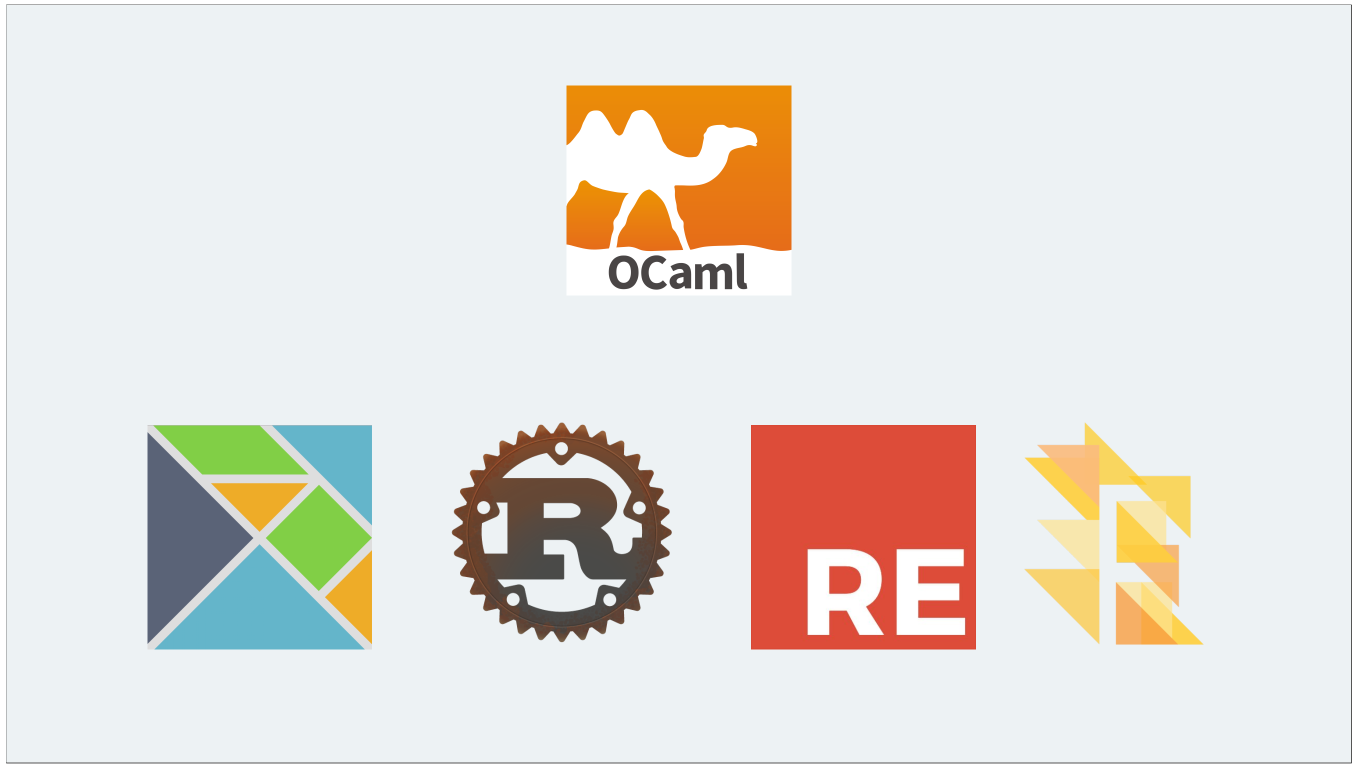


They were all influenced by OCaml.   2 decades

Rust and especially Elm both claim OCaml as an influence.
The first compiler for Rust was written in OCaml before it became self-hosting
Flow is, itself, written in OCaml.
Reason is literally OCaml with a new syntax

Pattern matching in Rust was inspired by OCaml's pattern matching
And now we have a TC39 proposal to add similar functionality to javascript
The recent pipeline operator proposal, OCaml had that too

Now, I'm not a clever man
but when I see this many smart people talking about something.
I want to understand why.
So I started learning OCaml.



I started to see why it had been so influential

Now, I'm still learning, and I'm not ready to talk about that,
I haven't yet seen to world respun in golden threads of lambda calculus
And I still have no idea what a monad is

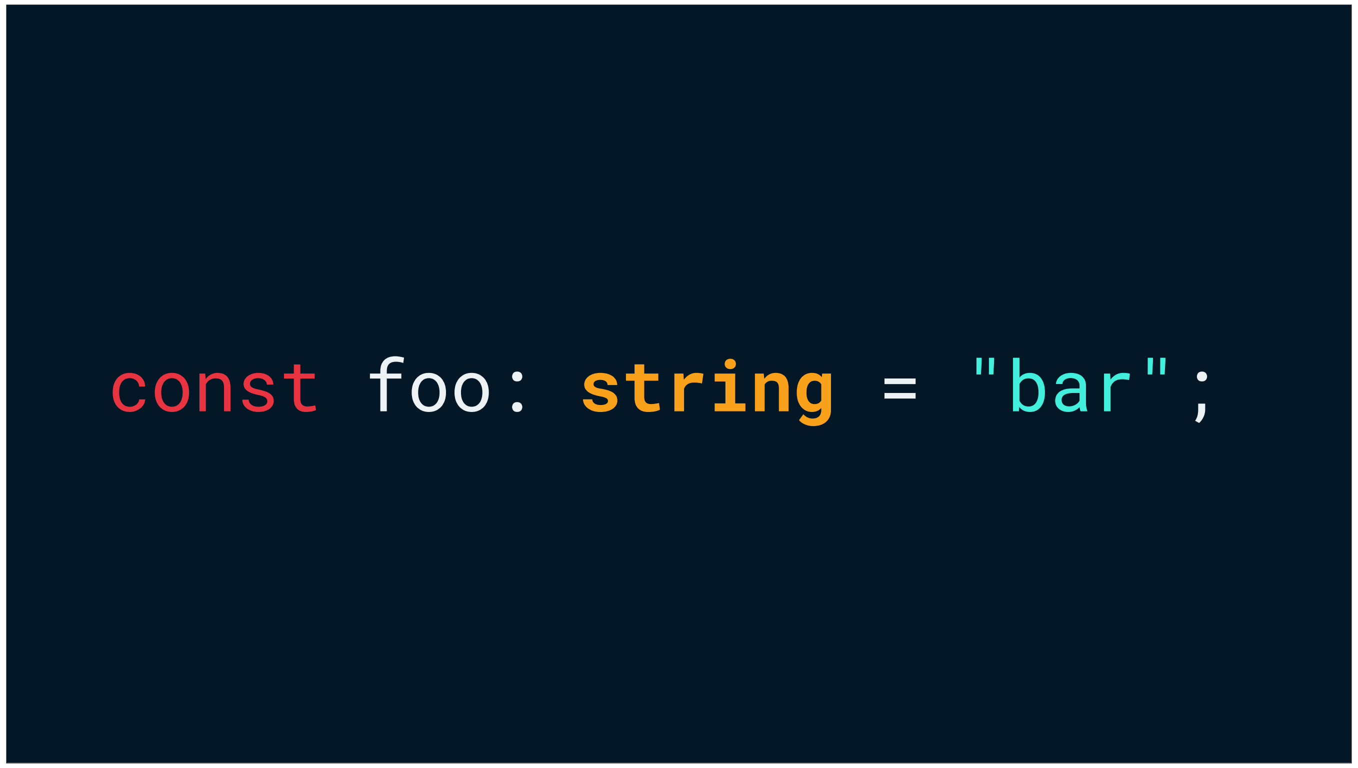
But what I found was that it started to change how I wrote JS

Flow supported enough of the OCaml idioms that
I started to write JavaScript with an OCaml accent

especially how I thought about leveraging types in my code

Minimum Viable Flow

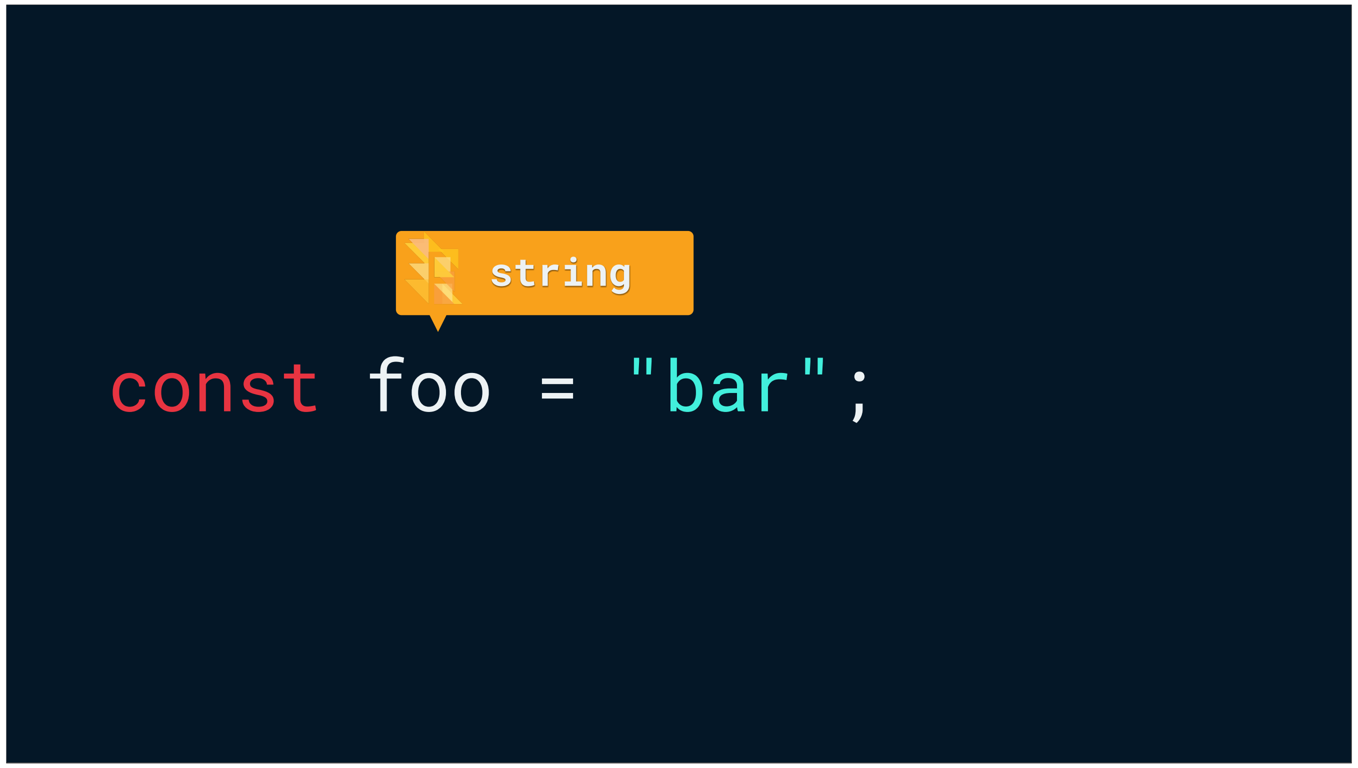
This talk isn't meant to be an intro to Flow, but if you haven't seen typed javascript before, let's go over a few things so that we're all on the same page



```
const foo: string = "bar";
```

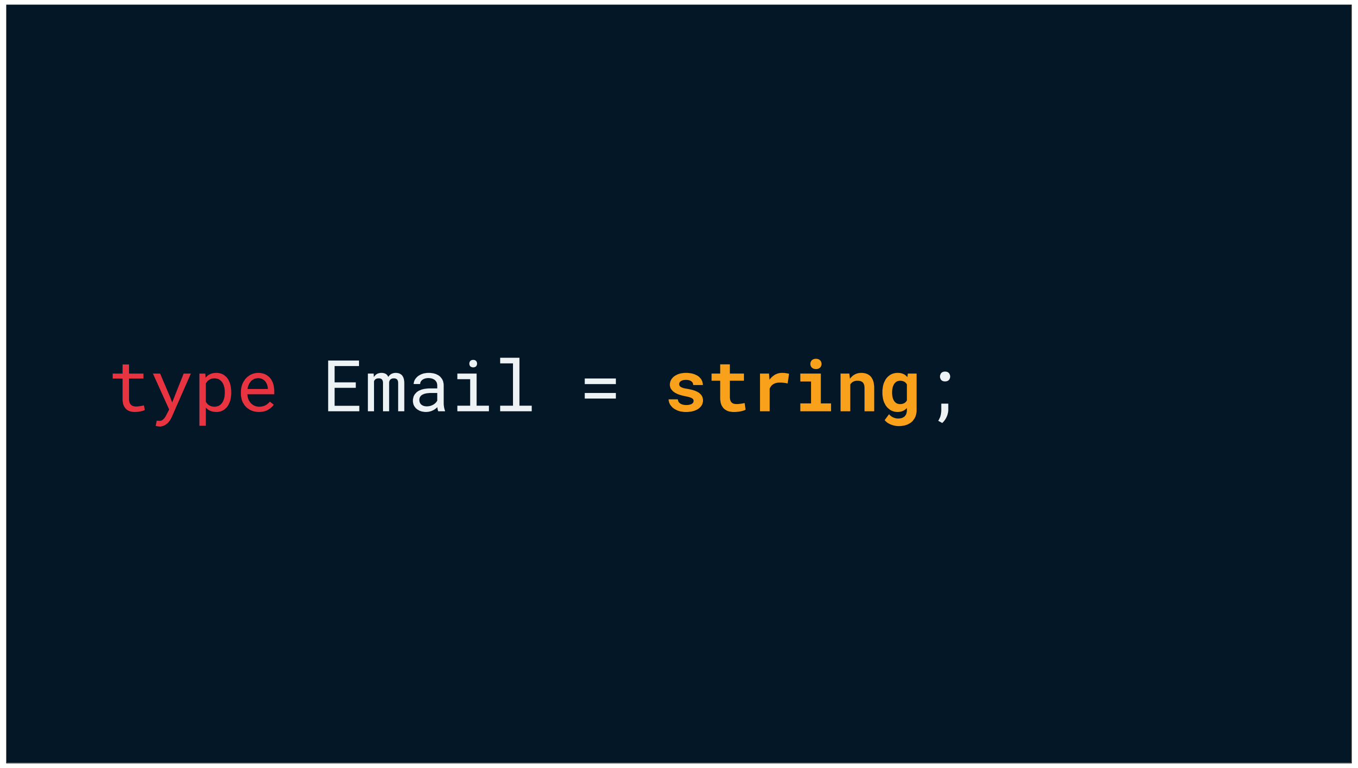
Allows you to annotate your code to define types

Here we say that foo is a string



```
string
const foo = "bar";
```

although modern type systems is also quite smart and can infer the type automatically in many cases



```
type Email = string;
```

You can give types a name. With this syntax

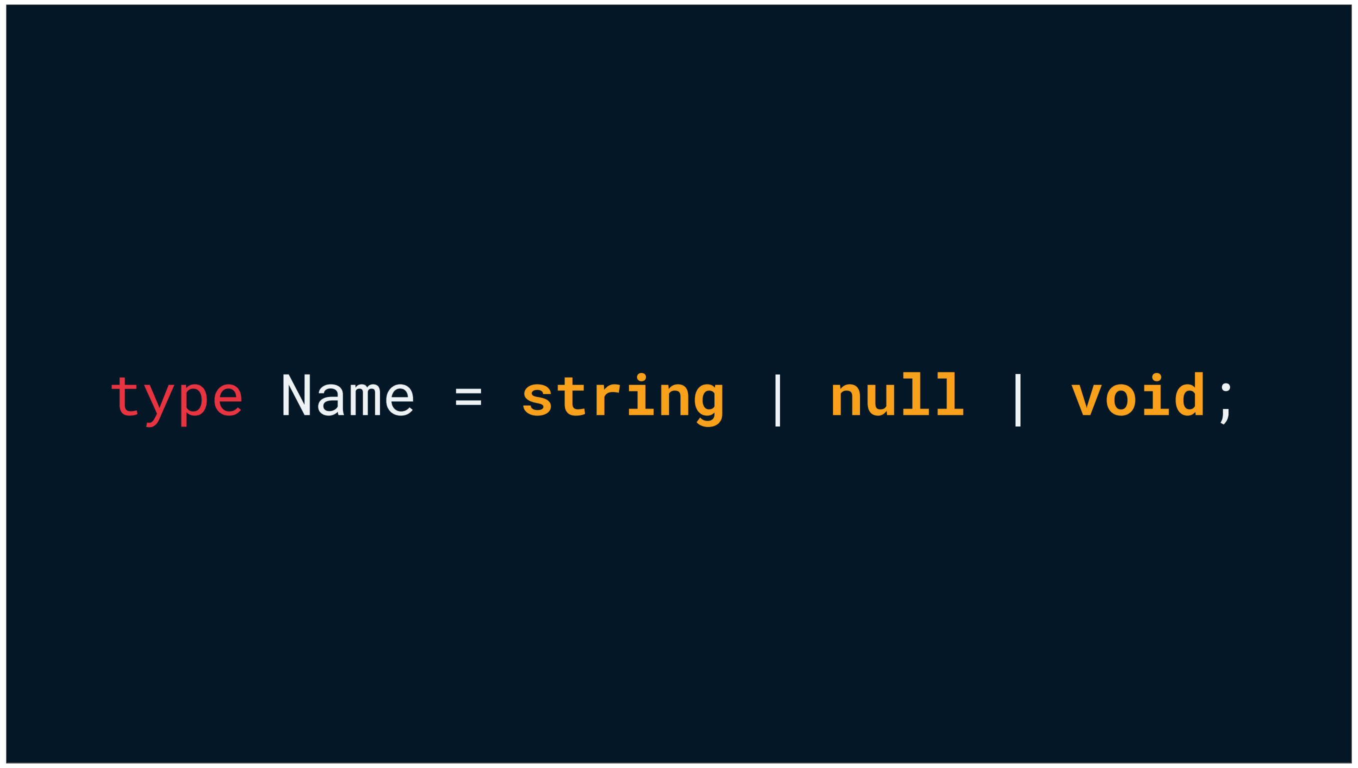
Here we’re aliasing string to convey some extra intent

```
function add(
  a: number,
  b: number
): number {
  return a + b;
}
```

This is the syntax for adding types to a function declaration

Due to inference Mostly you add type definitions to function definitions

Most of my JS code that uses types has type definitions at the top
And some types in the function declaration, but that’s it

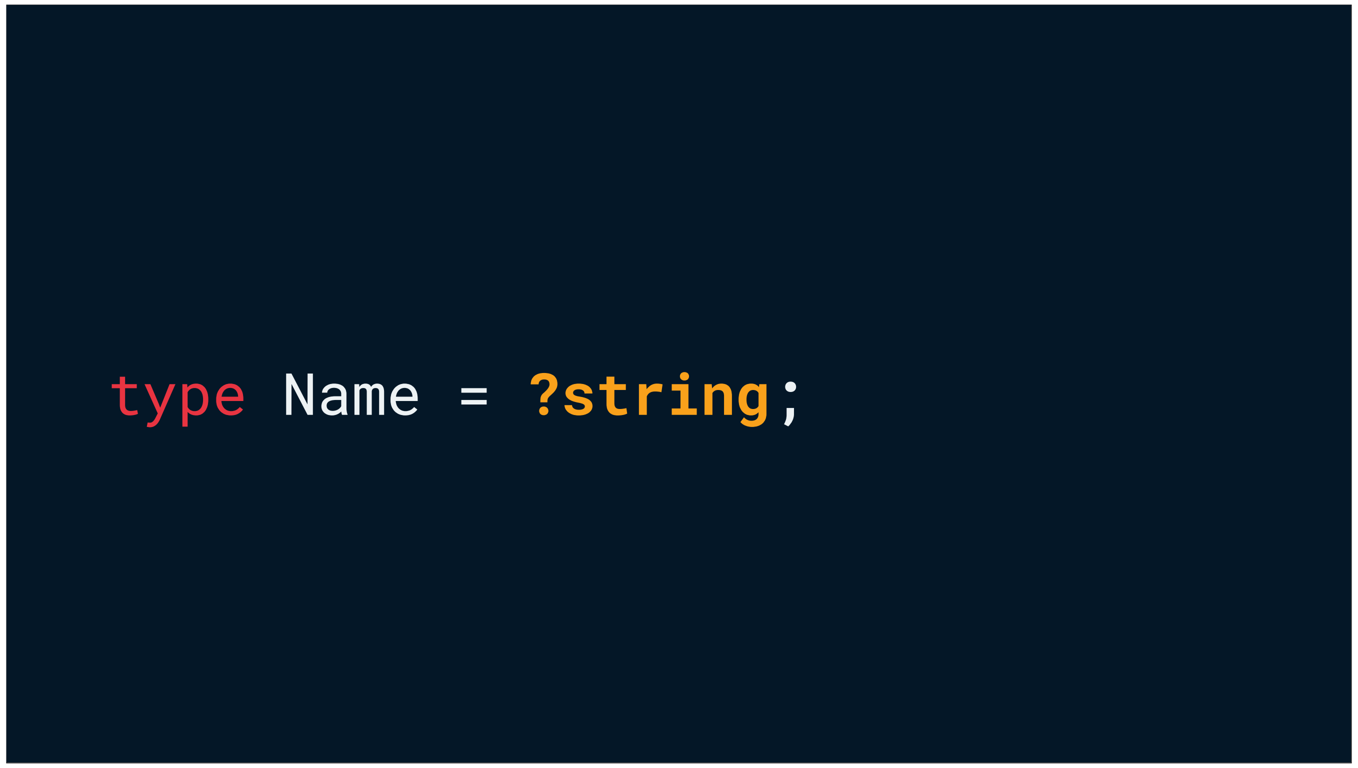


```
type Name = string | null | void;
```

You can create union types like this

You can read “void” as “undefined” here

This might be Ex: a type for an optional argument to a function



```
type Name = ?string;
```

It’s so common in the language that there’s a shorthand. This is the same as the previous slide.

We’d call this a “nullable string”

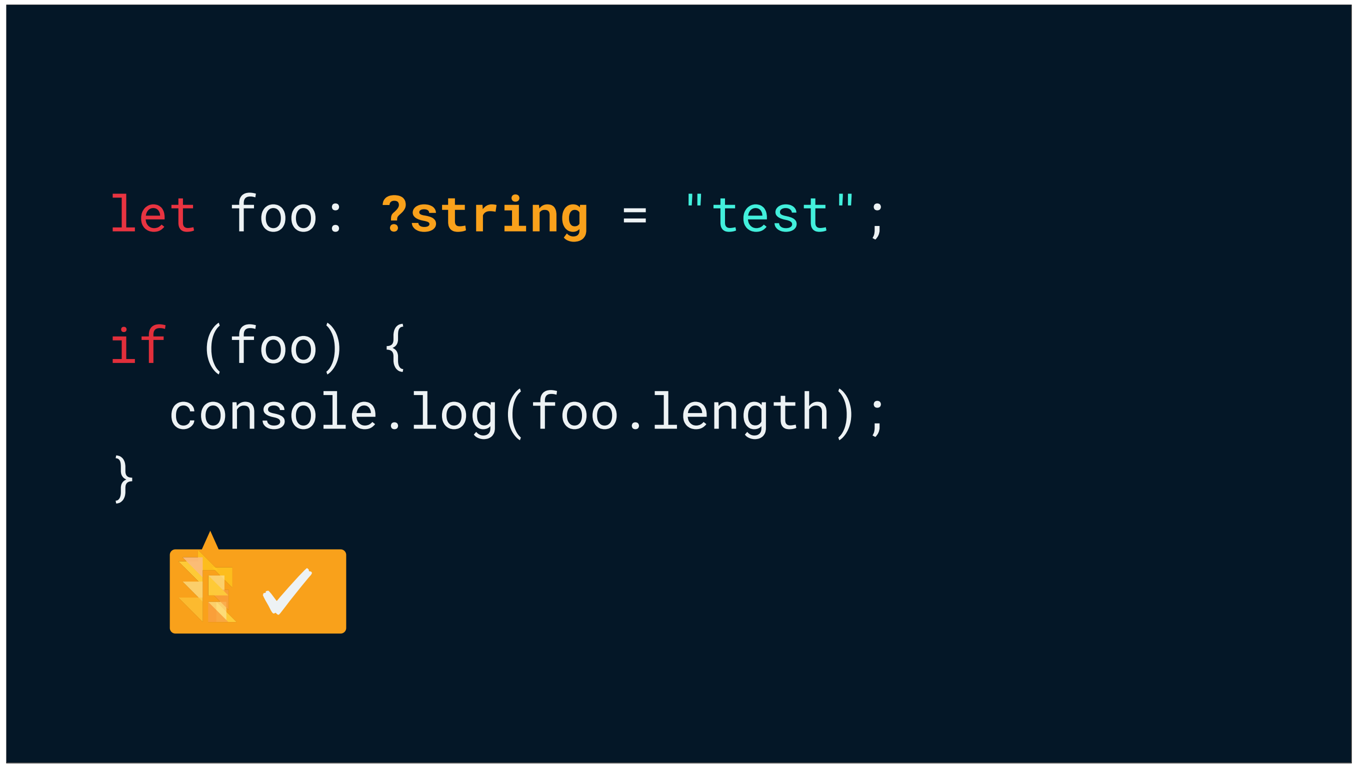
```
let foo: ?string = "test";

console.log(foo.length);
```

Union types kind of exist in this quantum state where they can be any of the types your unifying

We can do things common to all the types

If we try to treat a nullable string as a string, the type checker will let us know that this isn’t safe

If this were undefined, accessing length would blow up



```
let foo: ?string = "test";

if (foo) {
  console.log(foo.length);
}
```

✓

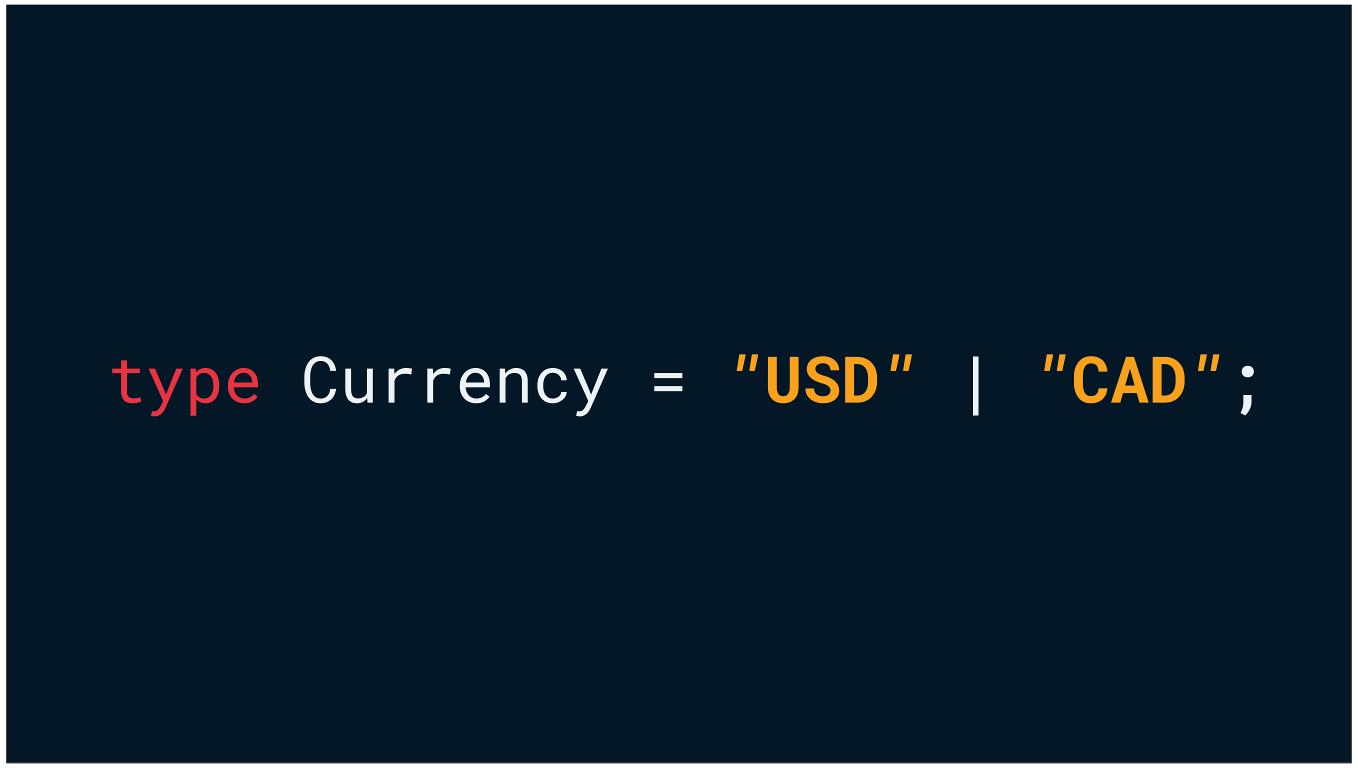
But if we check to see if foo is truthy, then flow can determine that inside the block

Foo must be a string, since null and undefined wouldn't pass the test

This is called type refinement

When people refer to Elm as "having no runtime" exceptions,
that doesn't come for free

Part of it is that the language will force you to handle these cases



```
type Currency = "USD" | "CAD";
```

Instead of saying something is a string
We can declare constant values as a type to create an enum

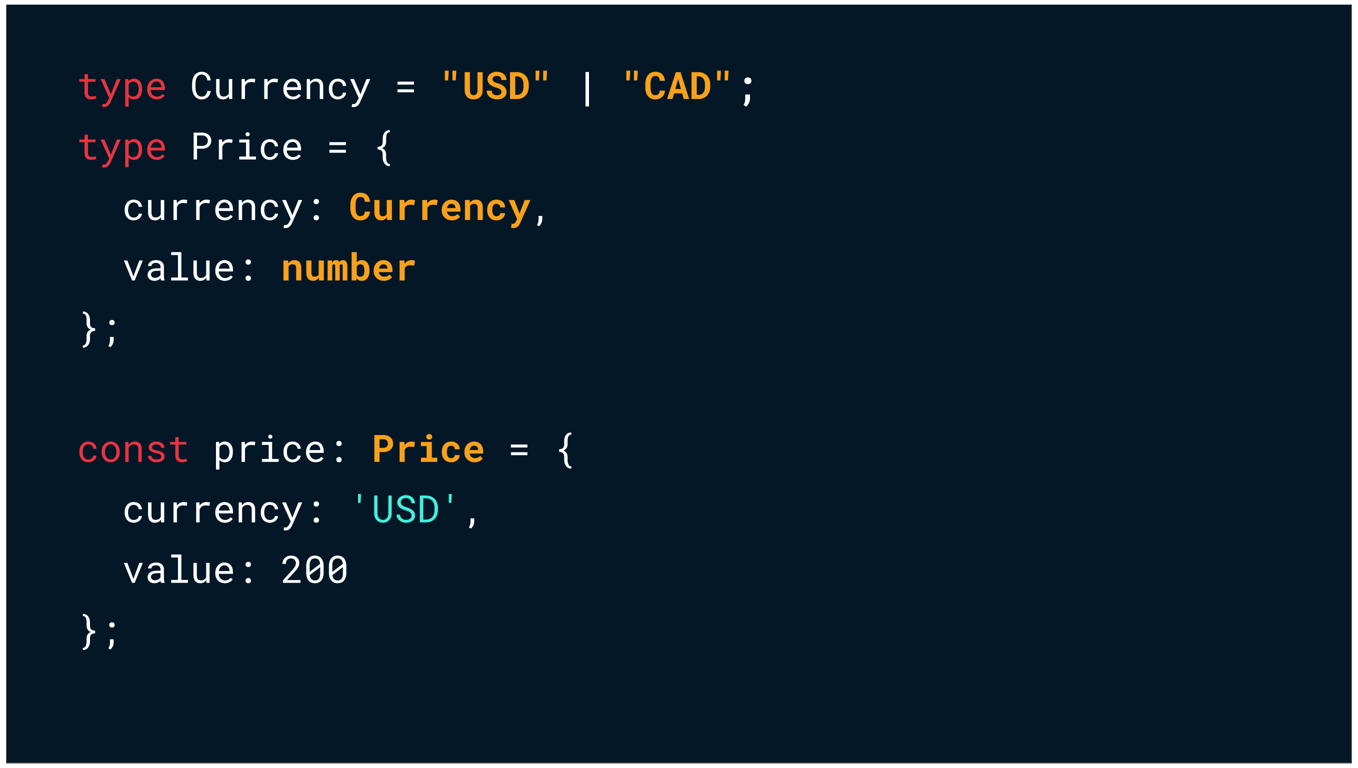
Anything with this type can *only* be one of these two values.

This is an example of a type that can’t be inferred

```
type Currency = "USD" | "CAD";
type Price = Object;




const price: Price = {
  currency: 'USD',
  value: 200
};
```

There is an Object type, but if you use it, the type system will know nothing about the contents

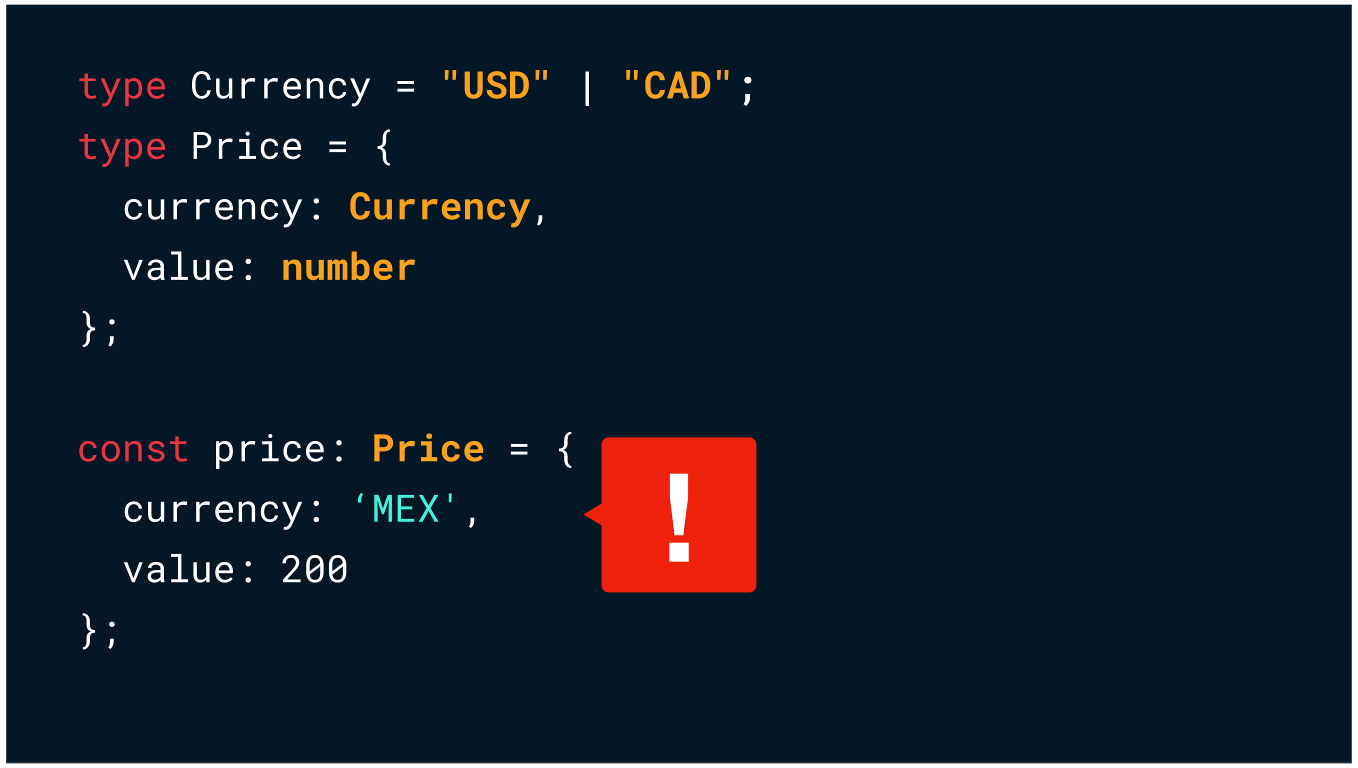
You can put anything in, take anything out.



```
type Currency = "USD" | "CAD";
type Price = {
  currency: Currency,
  value: number
};

const price: Price = {
  currency: 'USD',
  value: 200
};
```

The type system becomes much more useful when you start to define these structures

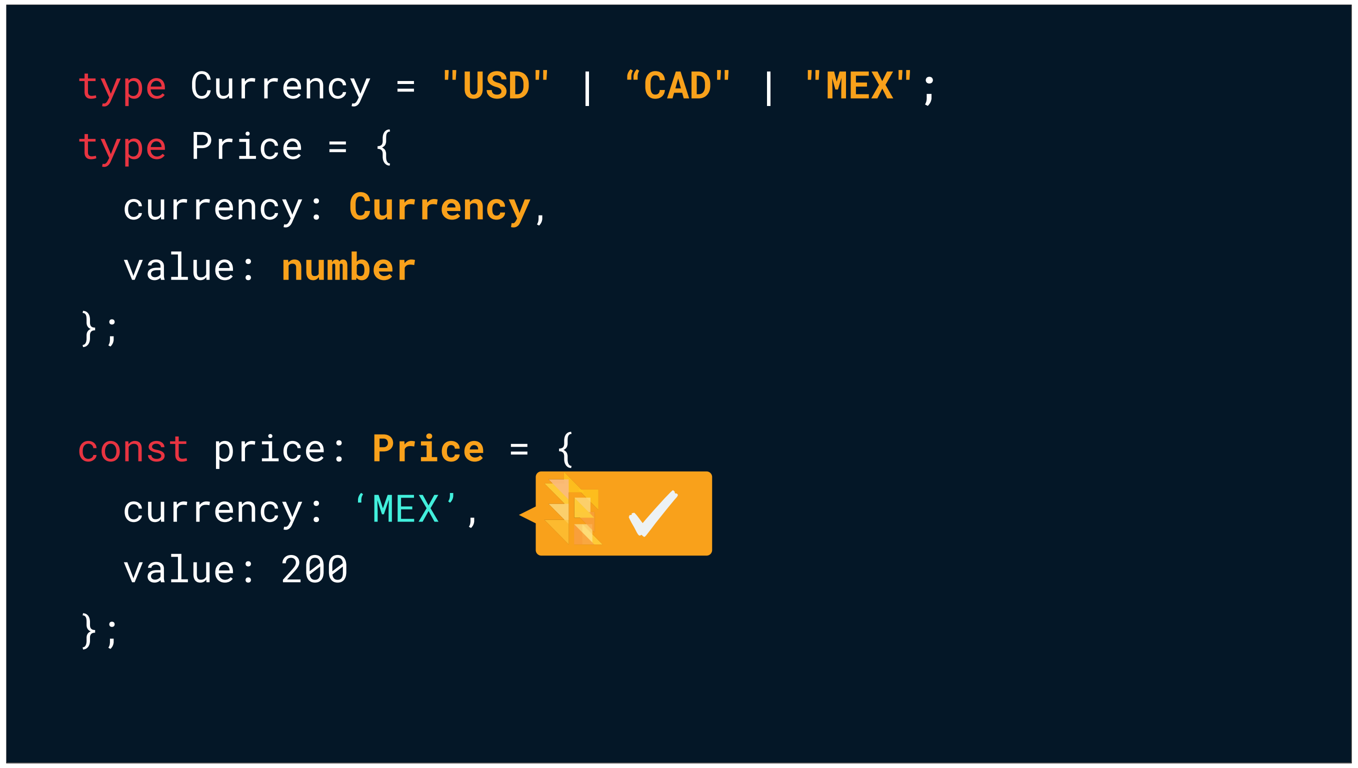
Value must be a number, it cannot be missing
For Currency, *only* one of these two strings is valid



```
type Currency = "USD" | "CAD";
type Price = {
  currency: Currency,
  value: number
};

const price: Price = {
  currency: 'MEX',
  value: 200
};
```

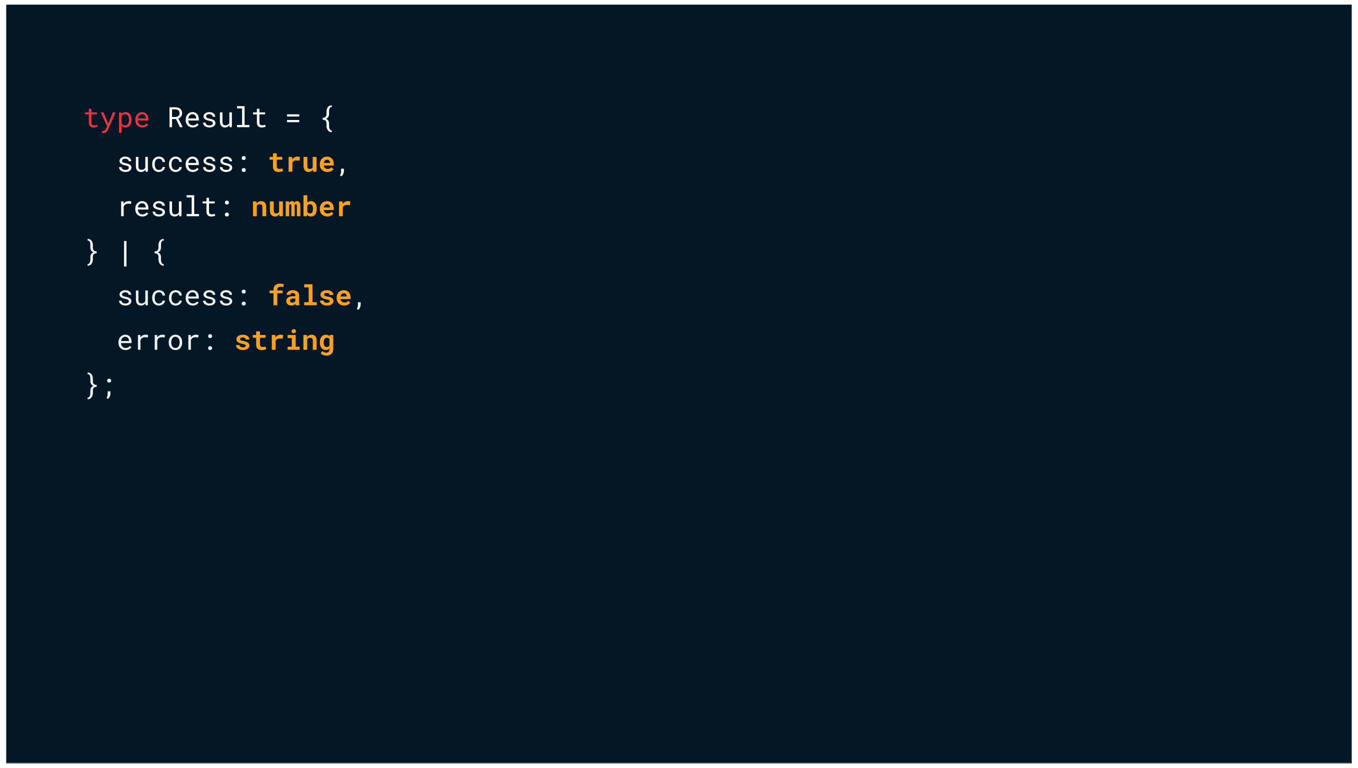
And if you ever assign something that doesn’t match the type, the type checker will catch it

Here someone decided that we supported pesos before we’ve updated the type



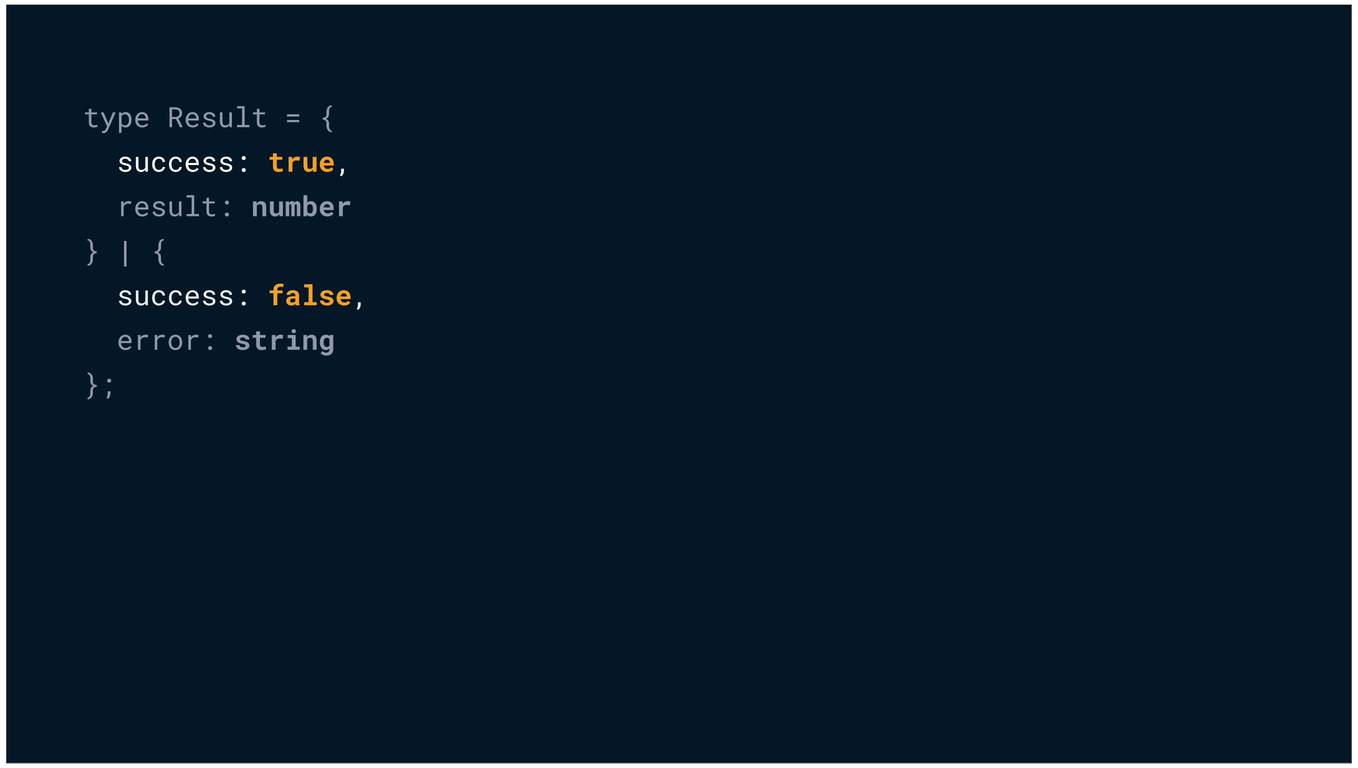
```
type Currency = "USD" | “CAD" | "MEX";
type Price = {
  currency: Currency,
  value: number
};

const price: Price = {
  currency: ‘MEX’,
  value: 200
};
```

If we add a new value to the enum we’re good



```
type Result = {
  success: true,
  result: number
} | {
  success: false,
  error: string
};
```

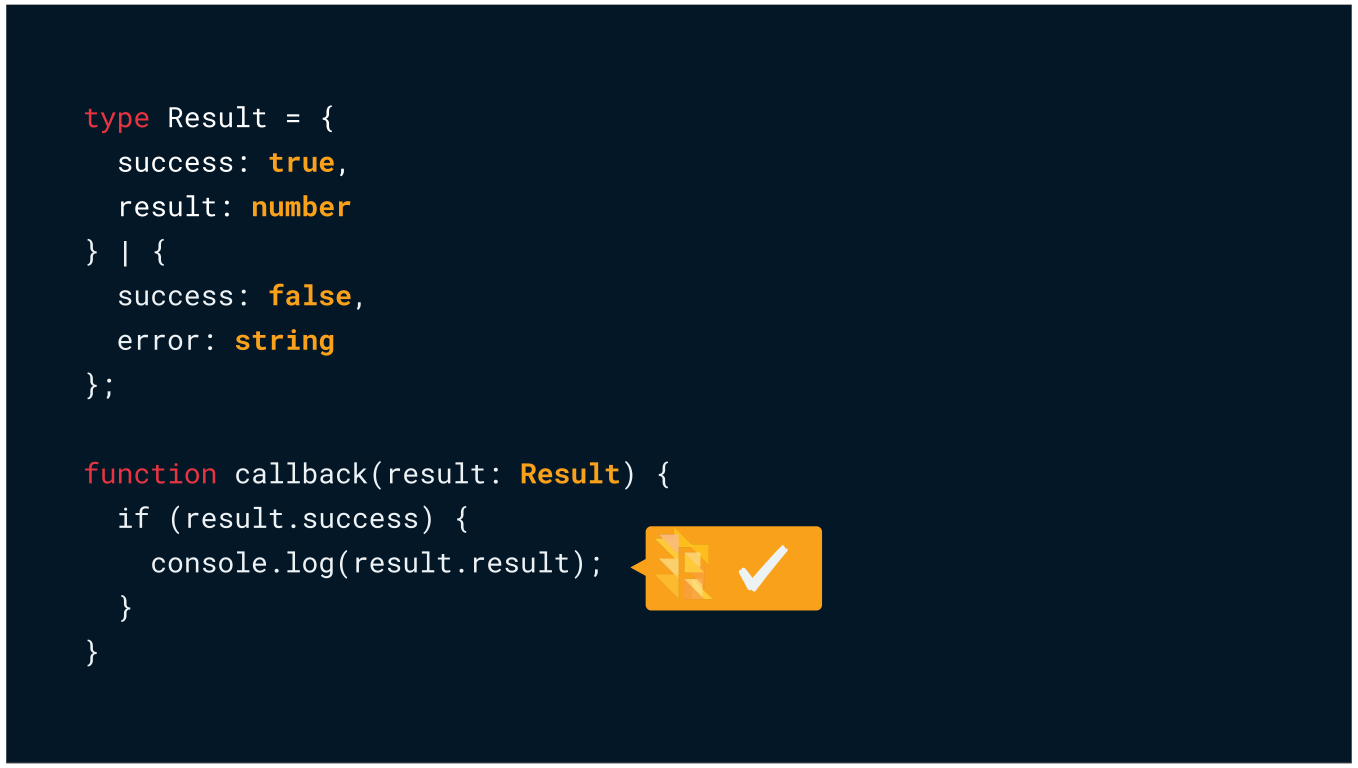
We can even compose the two features using something called a “disjoint union”



```
type Result = {
  success: true,
  result: number
} | {
  success: false,
  error: string
};
```

A disjoint union requires that all the shapes we're deciding between

contain a common property that flow can use to distinguish them, usually a specific string value to name it "tag"
Here we're using a boolean

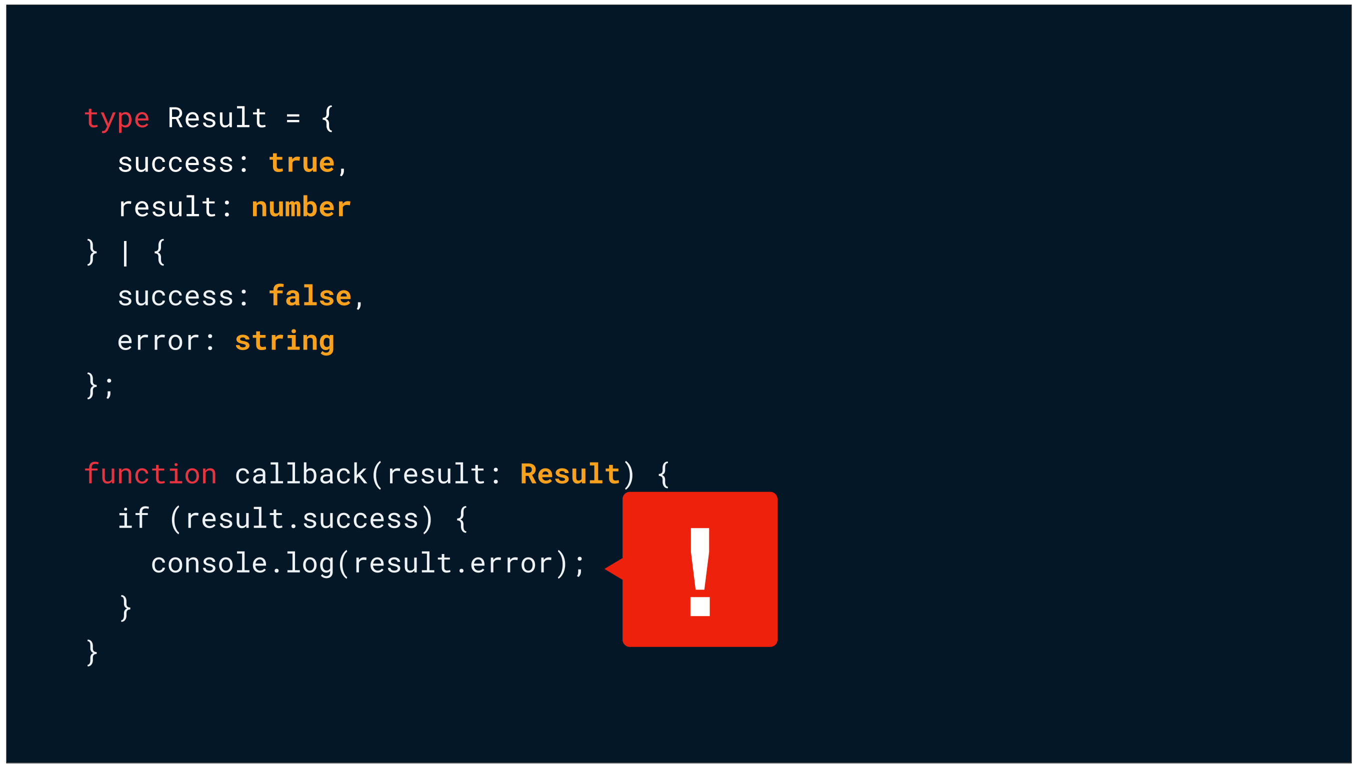


```
type Result = {
  success: true,
  result: number
} | {
  success: false,
  error: string
};

function callback(result: Result) {
  if (result.success) {
    console.log(result.result);
  }
}
```

Flow can then use this tag to do refinement just the way we saw before

Within this callback fn result is in this same “quantum state”

Here, the type system knows that if success is true
Then there HAS to be a result



```
type Result = {
  success: true,
  result: number
} | {
  success: false,
  error: string
};

function callback(result: Result) {
  if (result.success) {
    console.log(result.error);
  }
}
```

More importantly, it knows that the error property isn't present within this block
And correctly raises an error

Due to the refinement

It's not important for this talk, but the concept we're imitating has a name
“Algebraic Data Types”

There’s a lot more to flow’s type system
And a lot more to typescripts

But these few features pack a big punch
When you know how to use them

Let’s see how this might play out in a concrete example

Let’s pretend you are building a store app in plain JavaScript

And as a cheap ploy to keep your attention, let’s say it’s a candy store

Because everyone loves candy

```
function calculatePrice(count, item) {













  return count * item.price;
}
```

Every item has a price, and you need to calculate how much that item costs given how many are being purchased

Simple enough #ShipIt

We know this won’t stay simple because of the spacing

```
function calculatePrice(count, item) {









  return count * item.price;
}
```

But we shouldn’t ever expect our software to stay simple,

because software doesn’t exist to be simple and elegant

It exists to serve a function, and it’s natural it will change and grow and morph over time
As the business evolves

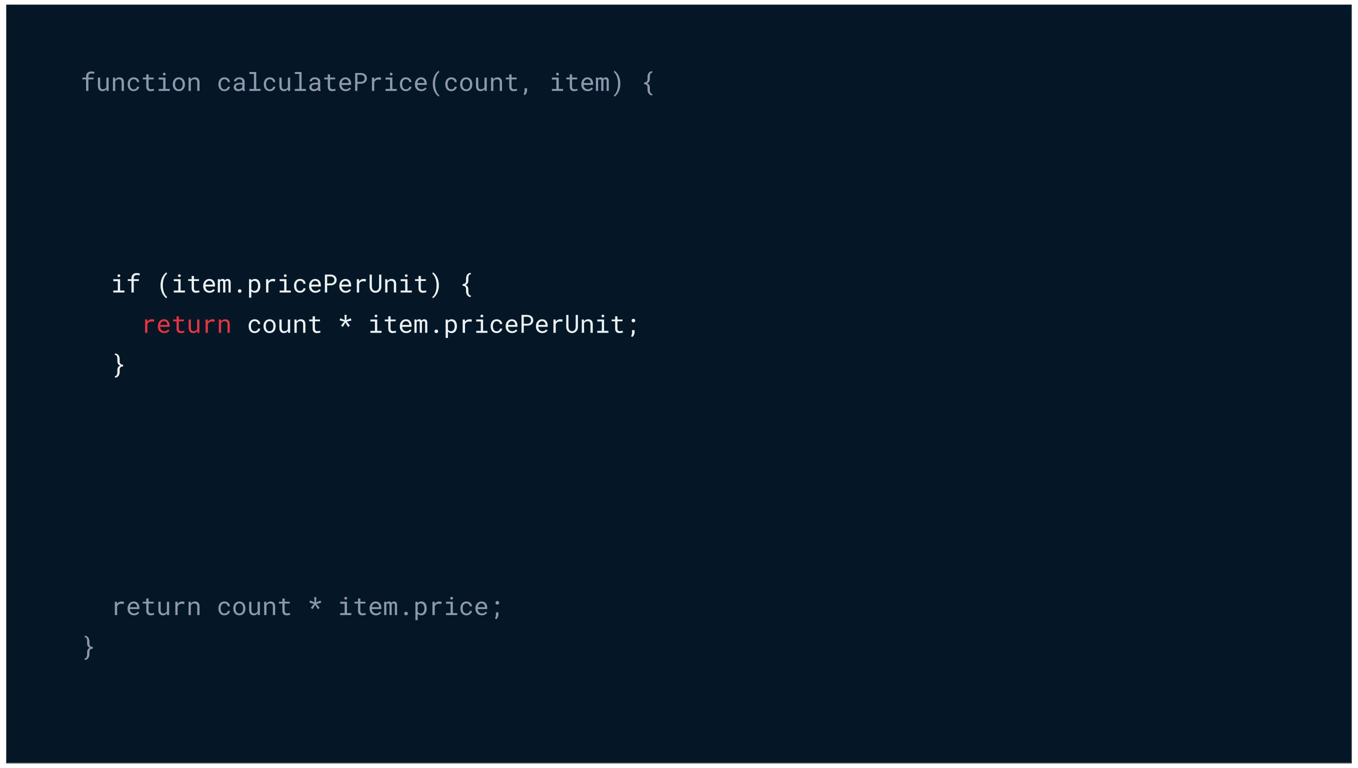
NB: You don’t have to try to read and understand the code over the next few slides.
I’ll point out what you need to know.

So we ship the first version of the store and it’s a great success
Everyone is walking down the sidewalk with big chocolate bars
It’s a beautiful thing

Then the owner of the store went to the airport and saw the candy store there

Their eyes were opened to the wonders of bulk bins! We have to have this!

This will take our store to the next level



```
function calculatePrice(count, item) {



  if (item.pricePerUnit) {
    return count * item.pricePerUnit;
  }





  return count * item.price;
}
```

So now an item either has a bulk price, or a per-item price,

that worked okay

This isn’t so bad

The store decides that they want to run a sale on some items

10% off lollipops!
20% off hersheys!

```
function calculatePrice(count, item) {




  if (item.pricePerUnit) {
    return count * item.pricePerUnit;
  } else if (item.discount) {
    return (1 - item.discount) * count * item.price;
  }




  return count * item.price;
}
```

So some items will have discounts now

That’s just one more case

The sale was a success, business is booming

so now we’re doing

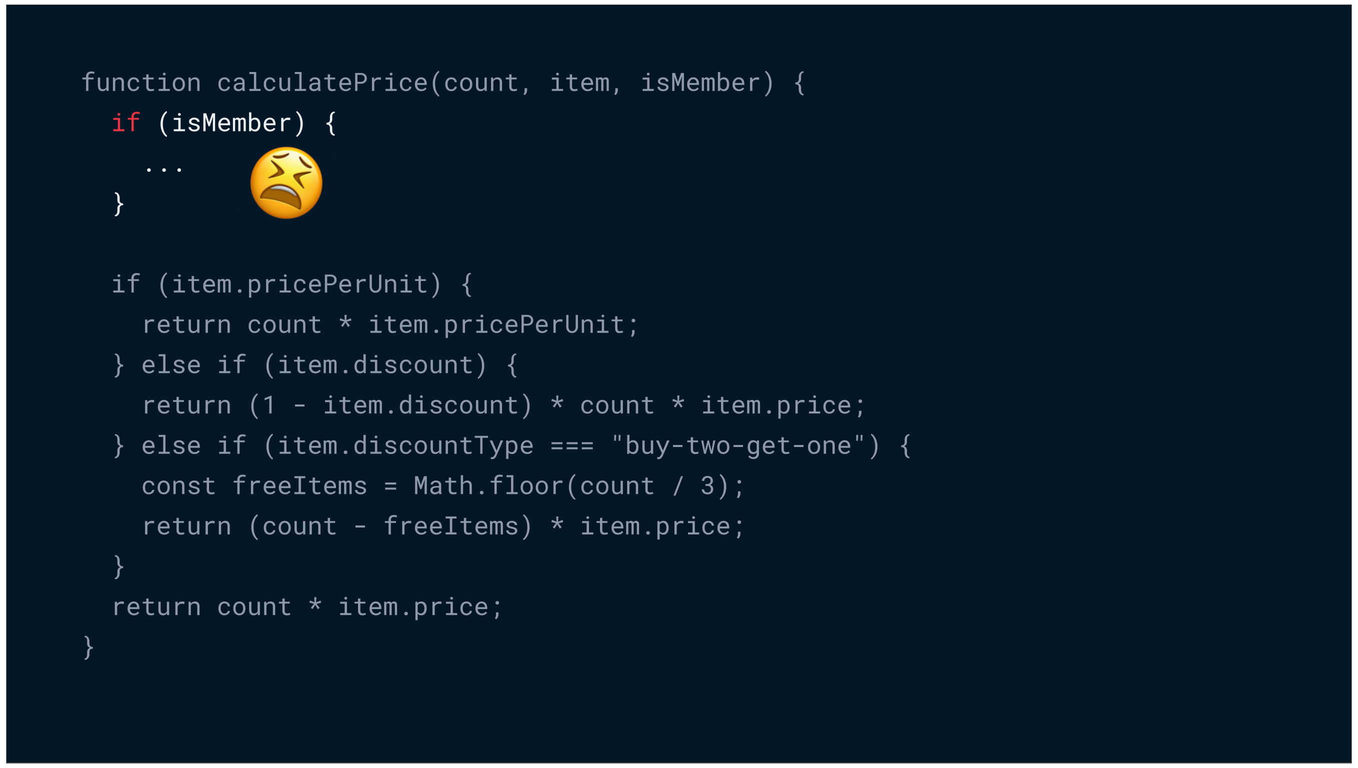
Buy Two get One free for rock candy

```
function calculatePrice(count, item) {




  if (item.pricePerUnit) {
    return count * item.pricePerUnit;
  } else if (item.discount) {
    return (1 - item.discount) * count * item.price;
  } else if (item.discountType === "buy-two-get-one") {
    const freeItems = Math.floor(count / 3);
    return (count - freeItems) * item.price;
  }
  return count * item.price;
}
```

Another candy store opened up down the block, so we want to encourage people to continue to shop with us

Now we set lower prices on some items for members

Special prices on both items and bulk bins!



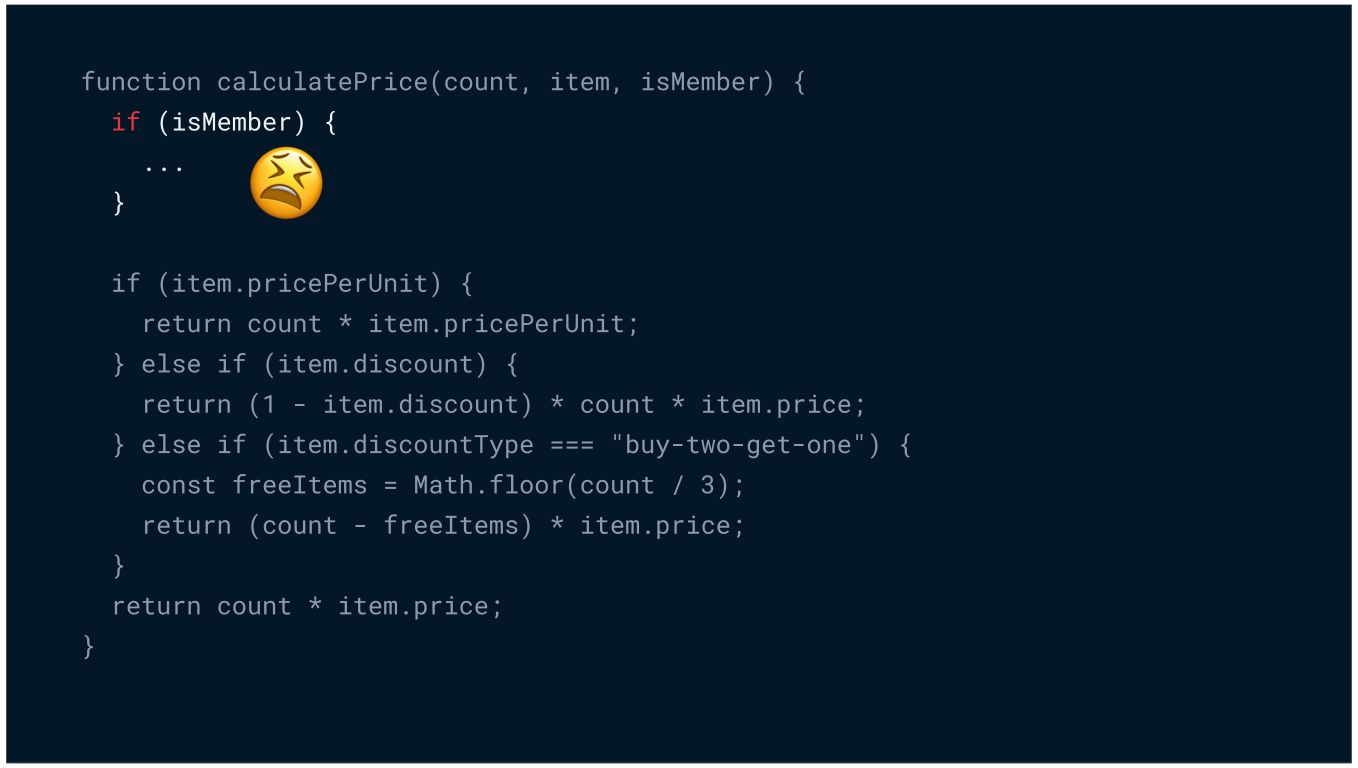
```
function calculatePrice(count, item, isMember) {
  if (isMember) {
    ...
  }

  if (item.pricePerUnit) {
    return count * item.pricePerUnit;
  } else if (item.discount) {
    return (1 - item.discount) * count * item.price;
  } else if (item.discountType === "buy-two-get-one") {
    const freeItems = Math.floor(count / 3);
    return (count - freeItems) * item.price;
  }
  return count * item.price;
}
```

This is when I gave up trying to make an example

We've had an explosion of complexity,
There's a good chance we'll end up applying a discount to the wrong thing
Or miscalculating a price
Or ignoring a member price

Worse, this object won't just be used in this function
It's going to be all over the code base
We need to know if an item has a discount to advertise that to non-members
We need to know if a special applies to an item to brand it differently
We are going to access these properties all over our code base, and it's all going to look like this
And it's getting harder and harder to make changes
More likely to lead to bugs

it's getting difficult to understand what is even in item



```
function calculatePrice(count, item, isMember) {
  if (isMember) {
    ...
  }

  if (item.pricePerUnit) {
    return count * item.pricePerUnit;
  } else if (item.discount) {
    return (1 - item.discount) * count * item.price;
  } else if (item.discountType === "buy-two-get-one") {
    const freeItems = Math.floor(count / 3);
    return (count - freeItems) * item.price;
  }
  return count * item.price;
}
```

And it won’t stop here:

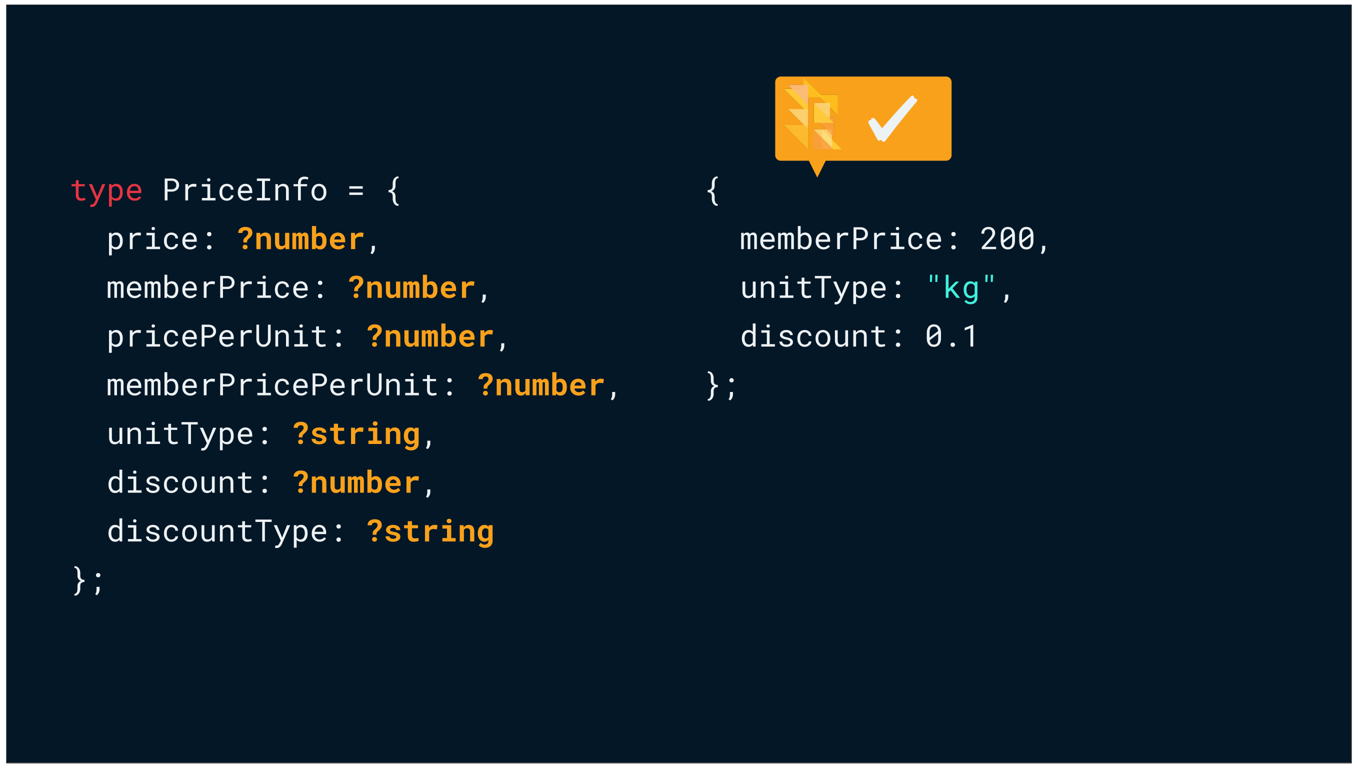
- You start to support multiple currencies
  - you add different levels of membership with different discounts
  - you offer discounts that apply to the cart as a whole: buy these two together and save X%
  - and so on

These are all reasonable features for a shop to have.
We need a better strategy to deal with this complexity

```
type PriceInfo = {
  price: ?number,
  memberPrice: ?number,
  pricePerUnit: ?number,
  memberPricePerUnit: ?number,
  unitType: ?string,
  discount: ?number,
  discountType: ?string
};
```

Looking at the type for our item

Every single property is optional

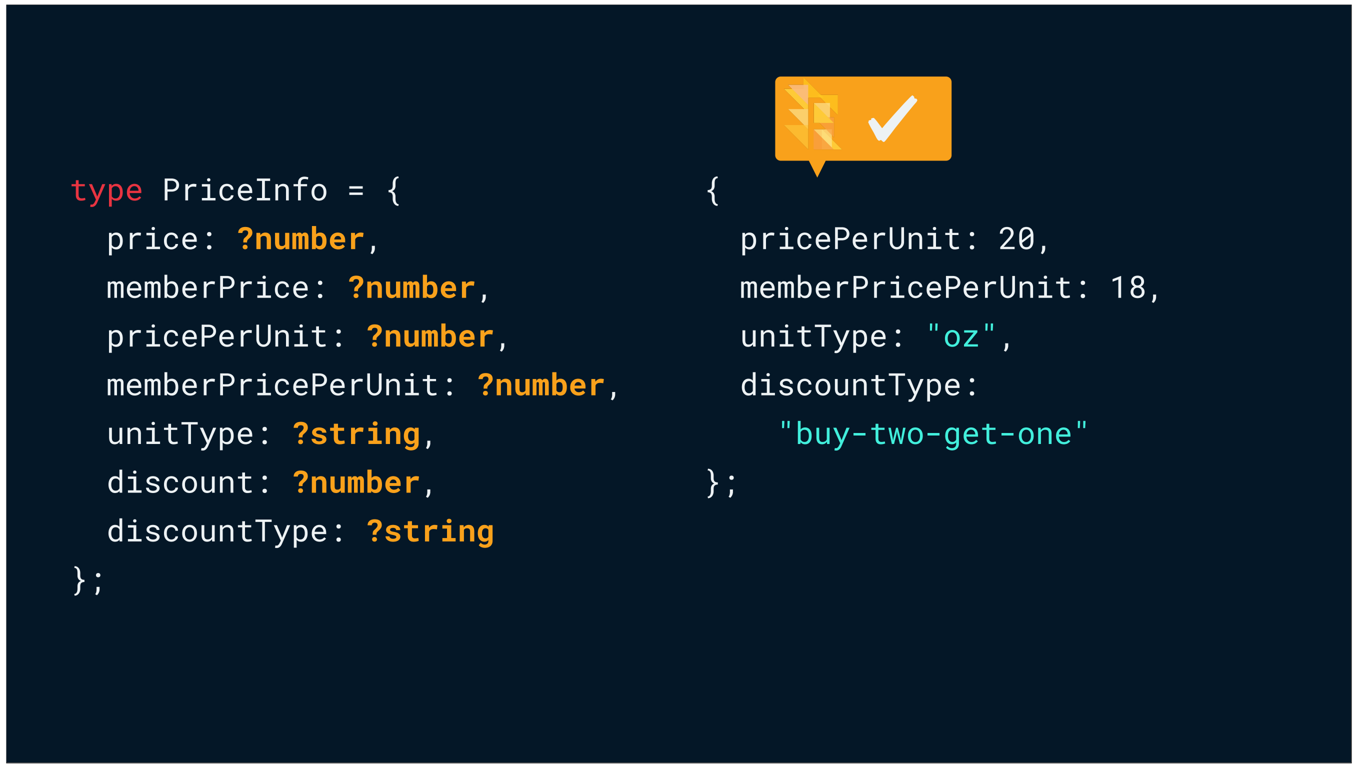


```
type PriceInfo = {
  price: ?number,
  memberPrice: ?number,
  pricePerUnit: ?number,
  memberPricePerUnit: ?number,
  unitType: ?string,
  discount: ?number,
  discountType: ?string
};
```

```
{
  memberPrice: 200,
  unitType: "kg",
  discount: 0.1
};
```

We can have price info that doesn’t actually make sense,
Member price without a price? But it’s sold in kilograms?

and the type system is going to be okay with it

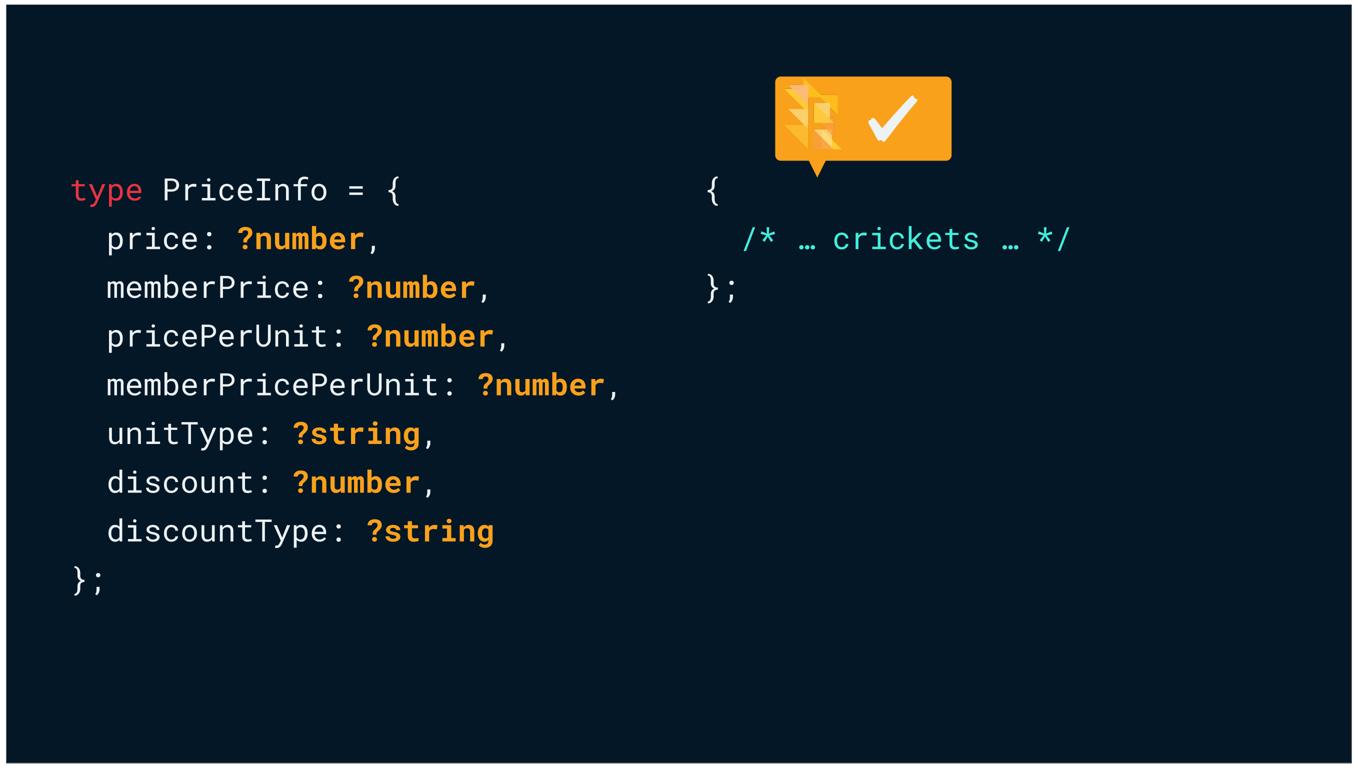


```
type PriceInfo = {                        {
  price: ?number,                           pricePerUnit: 20,
  memberPrice: ?number,                     memberPricePerUnit: 18,
  pricePerUnit: ?number,                    unitType: "oz",
  memberPricePerUnit: ?number,              discountType:
  unitType: ?string,                          "buy-two-get-one"
  discount: ?number,                      };
  discountType: ?string
};
```

What if we accidentally assign a discount where it doesn't make sense?

Does buy-two-get-one even make sense if we're talking about bulk bins?
Counting jelly beans

the Type system is okay with it though



```
type PriceInfo = {
  price: ?number,
  memberPrice: ?number,
  pricePerUnit: ?number,
  memberPricePerUnit: ?number,
  unitType: ?string,
  discount: ?number,
  discountType: ?string
};
```

```
{
  /* … crickets … */
};
```

Worse, since everything is optional even an empty object doesn't send up any red flags

How many times have you been debugging something, only to get to an object
That just doesn't make any sense?

We're forced to play computer in our heads
Or add a bunch of log statements
To try and figure out where this might have happened

In a language that forces you to refine nullable types
every time we access any property,
we're going to have to refine it,
meaning lots of null checks

Make Illegal States Impossible

Taking a cue from OCaml and other strongly typed languages
There’s a piece of advice for cases like this
Leverage the type system
So let’s explore what we can do here

```
type PriceInfo = {
  price: ?number,
  memberPrice: ?number,
  pricePerUnit: ?number,
  memberPricePerUnit: ?number,
  unitType: ?string,
  discount: ?number,
  discountType: ?string
};
```

There are a whole bunch of relationships between this data that we can’t see here

```
type PriceInfo = {
  price: ?number,
  memberPrice: ?number,
  pricePerUnit: ?number,
  memberPricePerUnit: ?number,
  unitType: ?string,
  discount: ?number,
  discountType: ?string
};
```

We shouldn’t have a member price without a non-member price

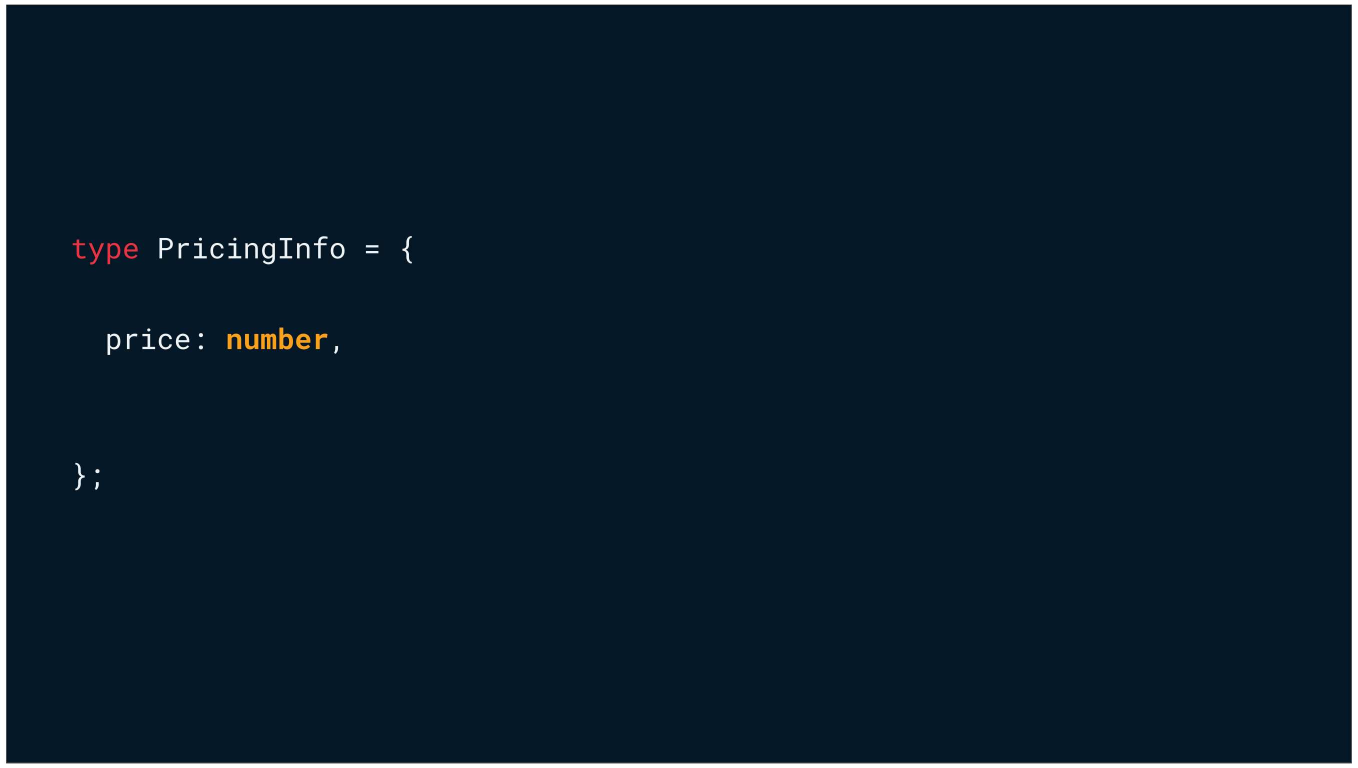
```
type PriceInfo = {
  price: ?number,
  memberPrice: ?number,
  pricePerUnit: ?number,
  memberPricePerUnit: ?number,
  unitType: ?string,
  discount: ?number,
  discountType: ?string
};
```

If we have a bulk price, we should know which unit we’re measuring

While we’re at it, we should probably enumerate the different unit types
Oz vs lbs

```
type PriceInfo = {
  price: ?number,
  memberPrice: ?number,
  pricePerUnit: ?number,
  memberPricePerUnit: ?number,
  unitType: ?string,
  discount: ?number,
  discountType: ?string
};
```

What discounts are there?
Can an item be both on-sale and priced at buy-two-get-one?



```
type PricingInfo = {
  price: number,

};
```

So we return to flow to try and sort this out

We start out with just a price

```
type PricingInfo = {
  type: 'single',
  price: number,


} | {
  type: 'bulk',
  pricePerUnit: number,
  unitType: 'oz' | ‘lb'

};
```

And now we have an extension point to add bulk bin pricing

By using a union we now have clear separation
It’s one or the other

We also define the units that we support

```
type PricingInfo = {
  type: 'single',
  price: number,
  memberPrice: ?number

} | {
  type: 'bulk',
  pricePerUnit: number,
  unitType: 'oz' | ‘lb',
  memberPricePerUnit: ?number
};
```

We can conveniently add membership prices

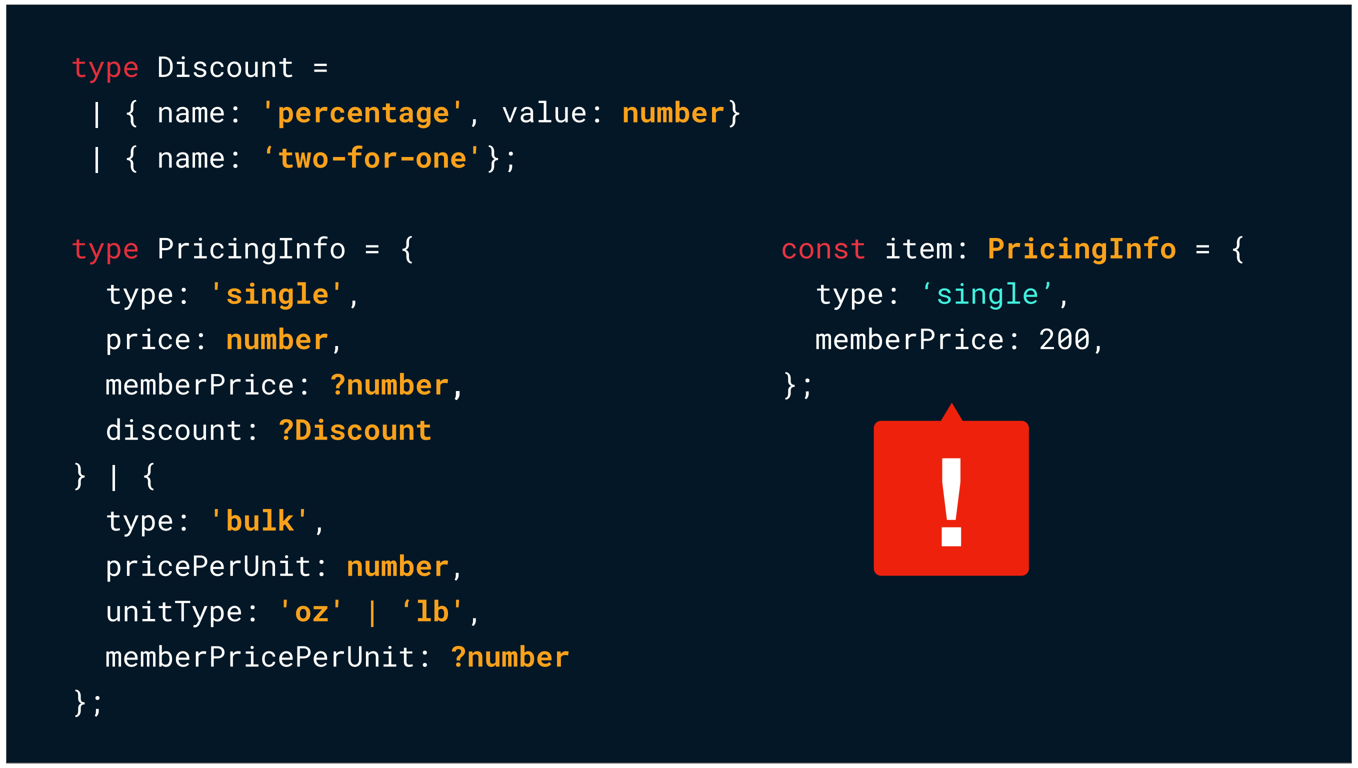
And now it’s clear that it may or may not be there

```
type Discount =
 | { name: 'percentage', value: number}
 | { name: ‘two-for-one'};

type PricingInfo = {
  type: 'single',
  price: number,
  memberPrice: ?number,
  discount: ?Discount
} | {
  type: 'bulk',
  pricePerUnit: number,
  unitType: 'oz' | ‘lb',
  memberPricePerUnit: ?number
};
```

And now we can enumerate the discounts we could offer, and what data each needs

It’s also become clear how we can add new discounts

Or if next week we start offering rentals of cotton candy machines for parties, we can add a “rental” type

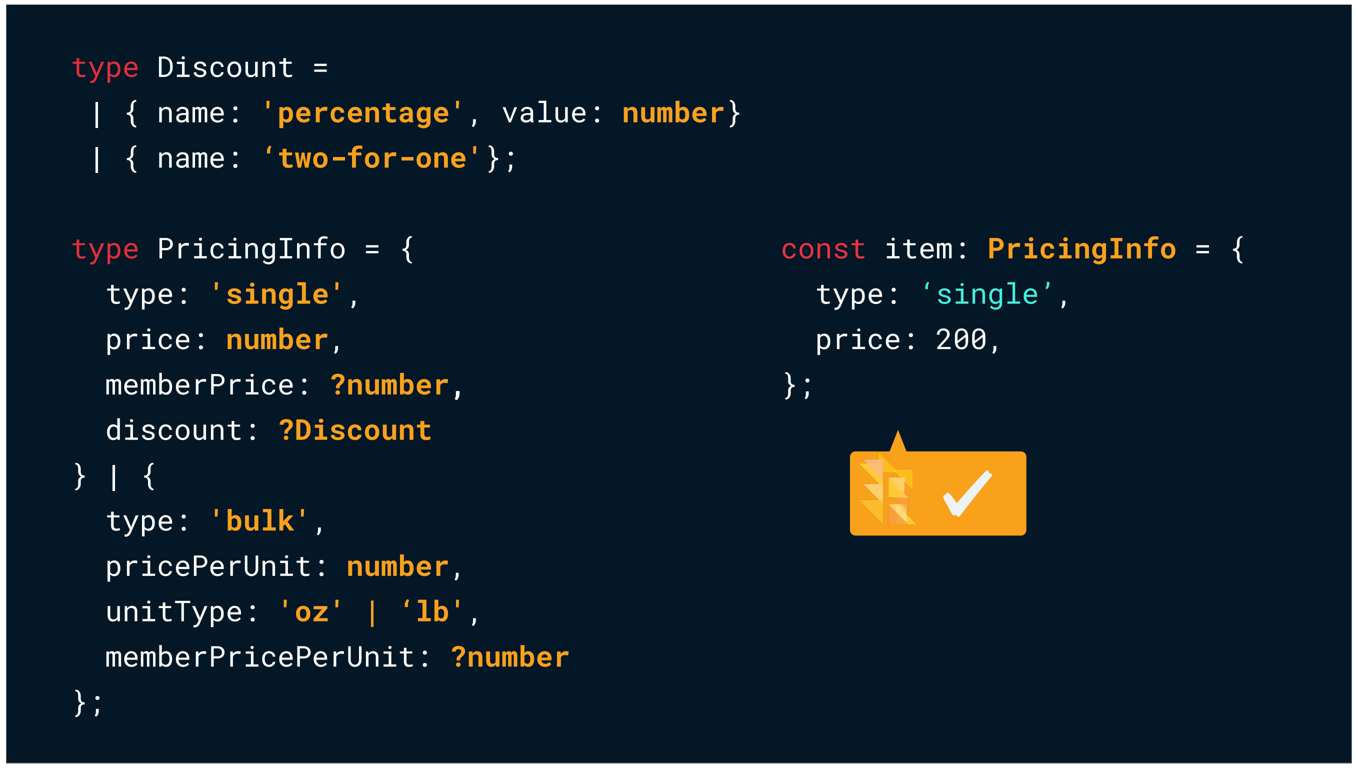


```
type Discount =
 | { name: 'percentage', value: number}
 | { name: ‘two-for-one'};

type PricingInfo = {
  type: 'single',
  price: number,
  memberPrice: ?number,
  discount: ?Discount
} | {
  type: 'bulk',
  pricePerUnit: number,
  unitType: 'oz' | ‘lb',
  memberPricePerUnit: ?number
};
```

```
const item: PricingInfo = {
  type: ‘single’,
  memberPrice: 200,
};
```

Best of all, if someone tries to build an object with some parameters that don’t make sense

The type system will catch it



```
type Discount =
 | { name: 'percentage', value: number}
 | { name: ‘two-for-one'};

type PricingInfo = {                          const item: PricingInfo = {
  type: 'single',                               type: ‘single’,
  price: number,                                price: 200,
  memberPrice: ?number,                       };
  discount: ?Discount
} | {
  type: 'bulk',
  pricePerUnit: number,
  unitType: 'oz' | ‘lb',
  memberPricePerUnit: ?number
};
```

It’s actually now impossible to represent a price info object that doesn’t match our business rules

And the additional structure means that writing a function like calculatePrice is a lot more straightforward

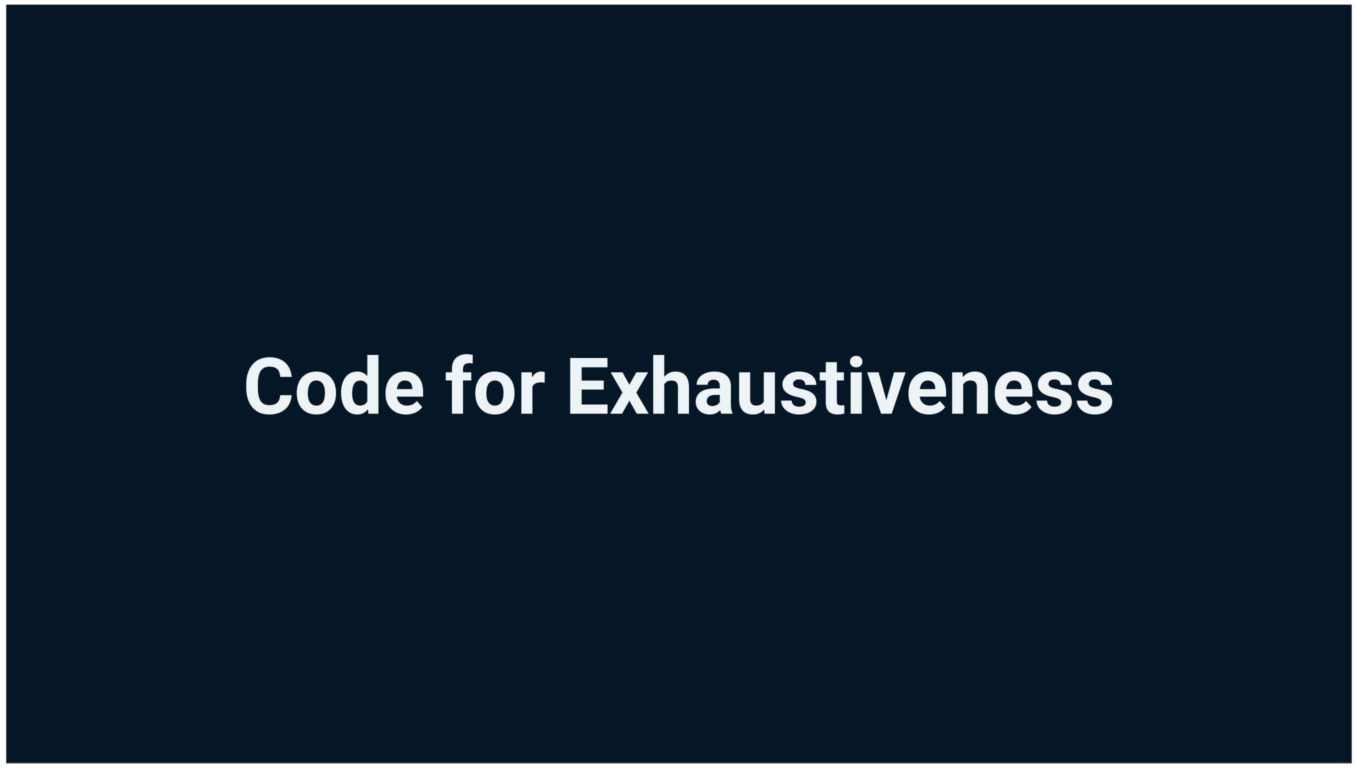
And if we mess up and try to apply a discount to a bulk item, flow is going to catch it

```
type Discount =
 | { name: 'percentage', value: number}
 | { name: ‘two-for-one'};

type PricingInfo = {
  type: 'single',
  price: number,
  memberPrice: ?number,
  discount: ?Discount
} | {
  type: 'bulk',
  pricePerUnit: number,
  unitType: 'oz' | ‘lb',
  memberPricePerUnit: ?number
};
```

We didn’t have to make any classes, or inherit anything

A little more than a dozens of lines of code

This serves as documentation
Converts intent
Enables the type system to do a lot for us

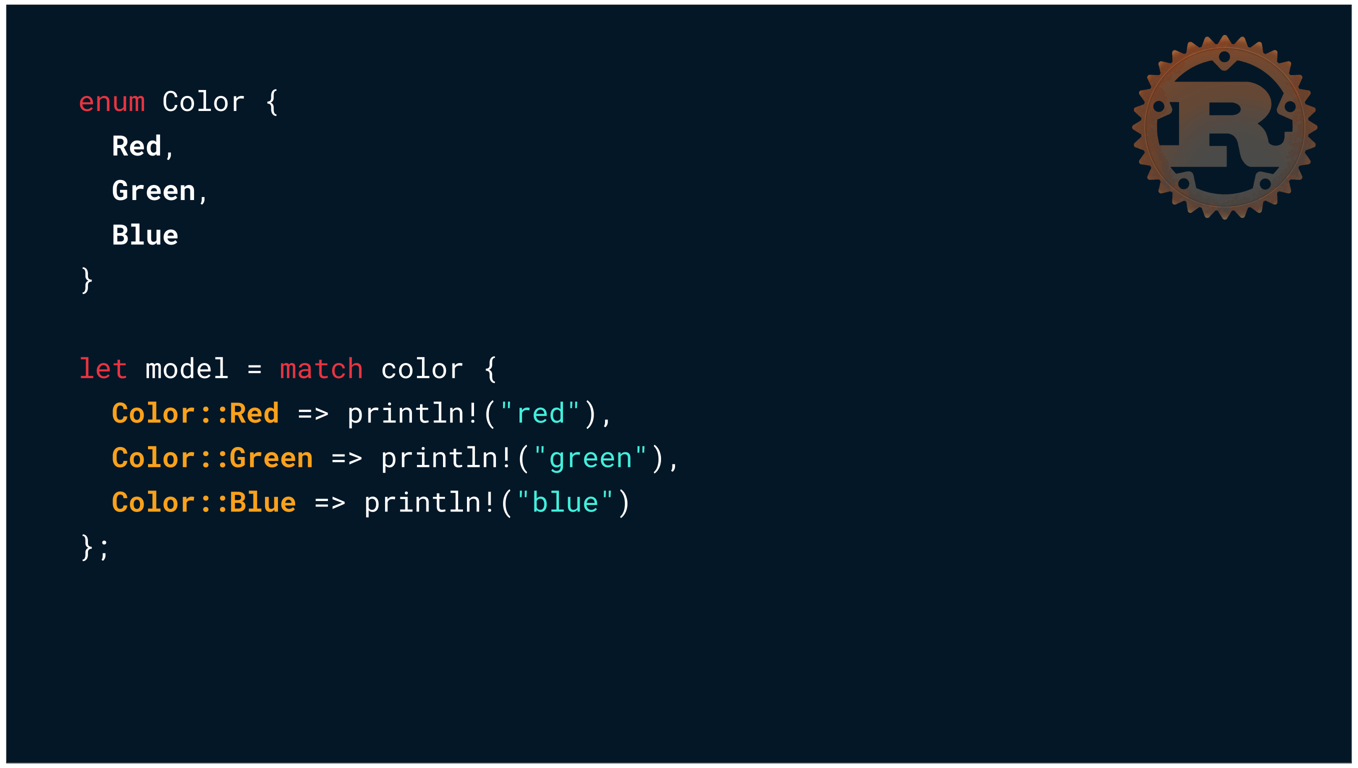


Code for Exhaustiveness

There's another idea that I want to talk about from these languages
I was first introduced to this idea when I started reading about Rust

But it was in OCaml before that, and who knows where before that

We should structure our code in such a way that
The type system can tell us when we're missing something



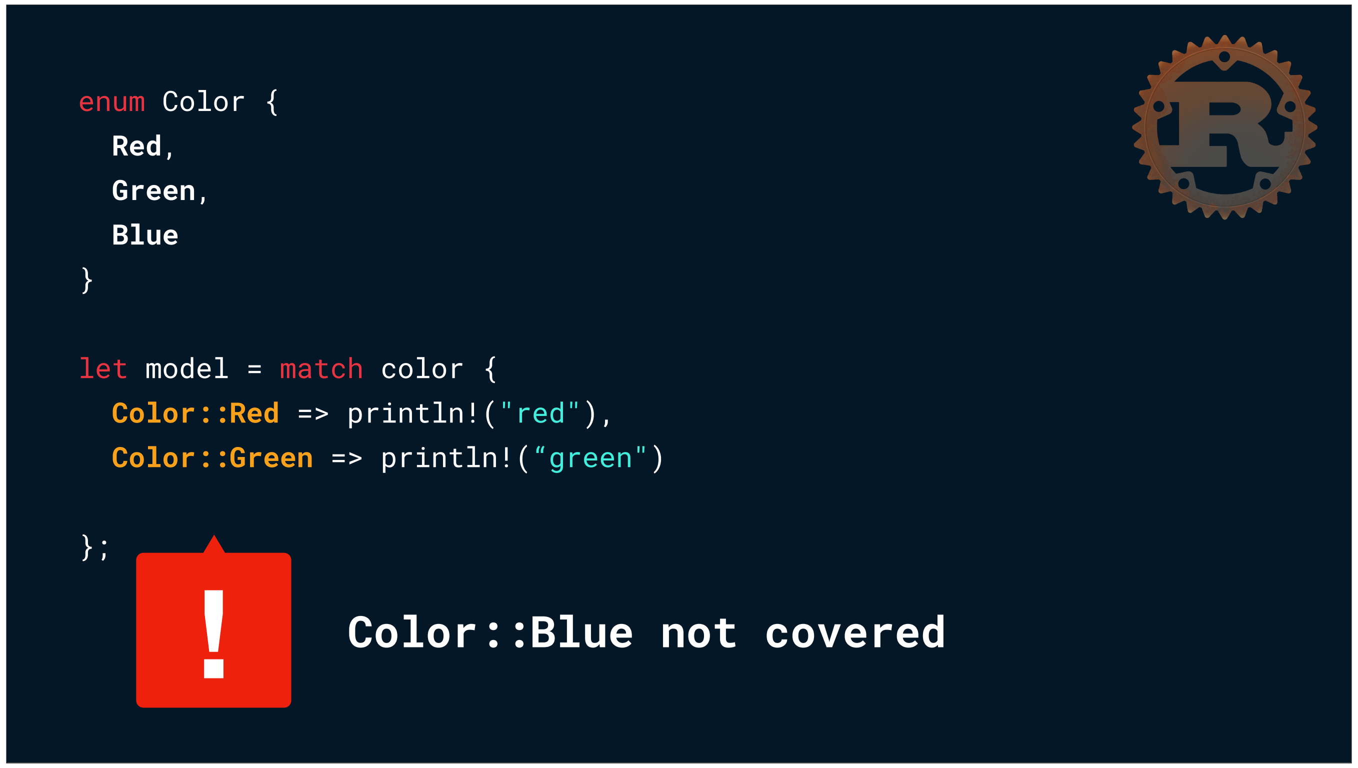
```rust
enum Color {
  Red,
  Green,
  Blue
}

let model = match color {
  Color::Red => println!("red"),
  Color::Green => println!("green"),
  Color::Blue => println!("blue")
};
```

In rust, match works kind of like a switch statement

Instead of using string values, Rust has enums in the language

And we can use match to branch on its value

an important difference



```
enum Color {
  Red,
  Green,
  Blue
}

let model = match color {
  Color::Red => println!("red"),
  Color::Green => println!("green")

};
```

```
Color::Blue not covered
```

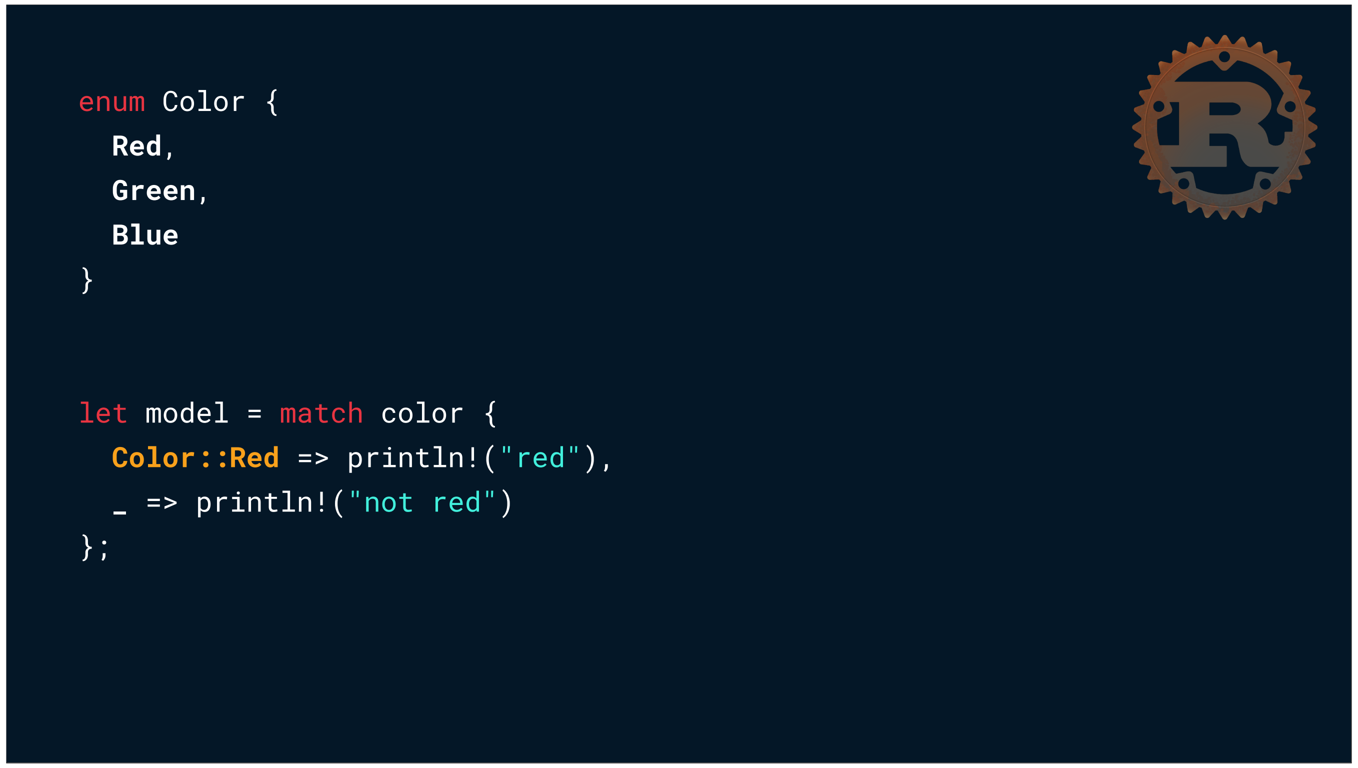
If the match isn't "exhaustive"

If we forget a possibility

Or if we add a new possibility to the enum later

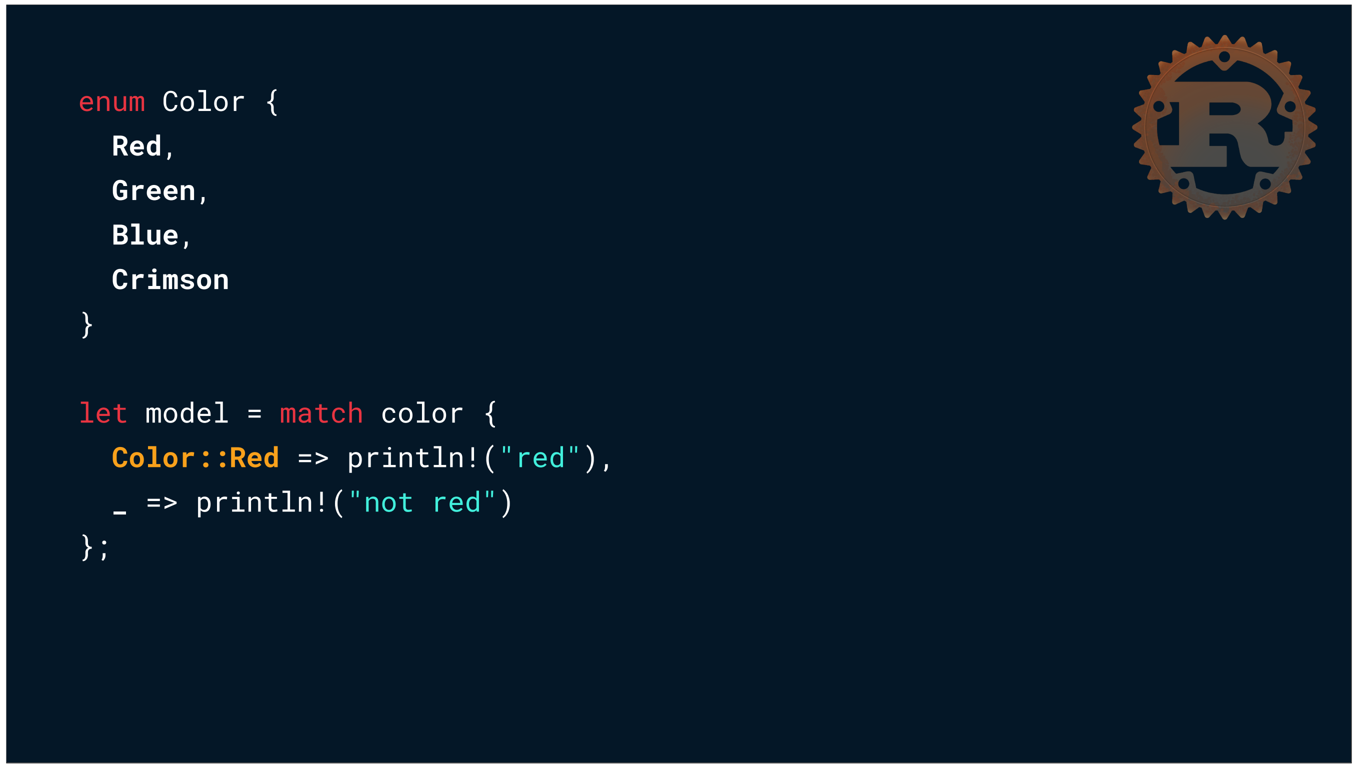
We'll get an error, and it will tell us exactly what case we're not covering

Elm has particularly good error messages for this


```
enum Color {
  Red,
  Green,
  Blue
}


let model = match color {
  Color::Red => println!("red"),
  _ => println!("not red")
};
```

We can make it harder to the type system to help us though

In rust _ in a match statement is like default in a switch statement

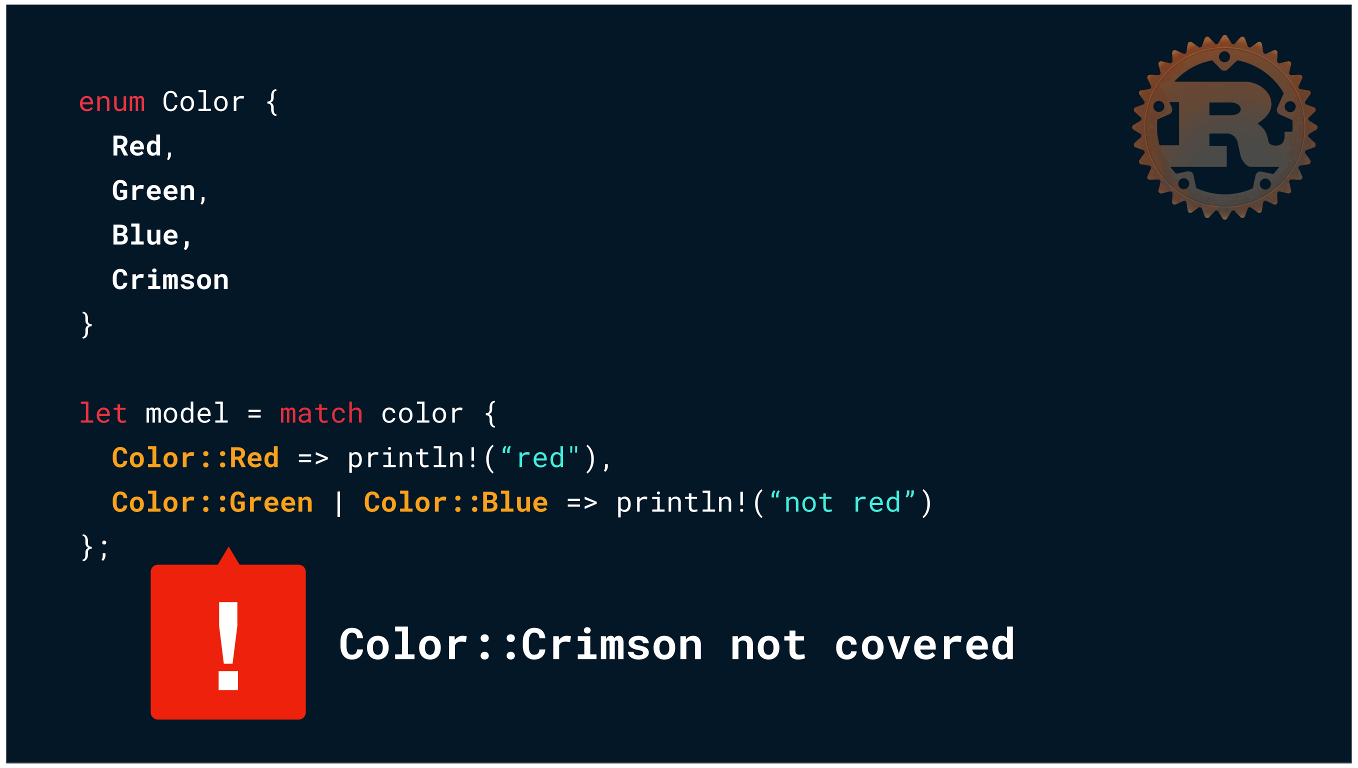


```
enum Color {
  Red,
  Green,
  Blue,
  Crimson
}

let model = match color {
  Color::Red => println!("red"),
  _ => println!("not red")
};
```

If we need to add another value to the enum

remember, programs change and evolve

Our type system is still going to think everything is okay here

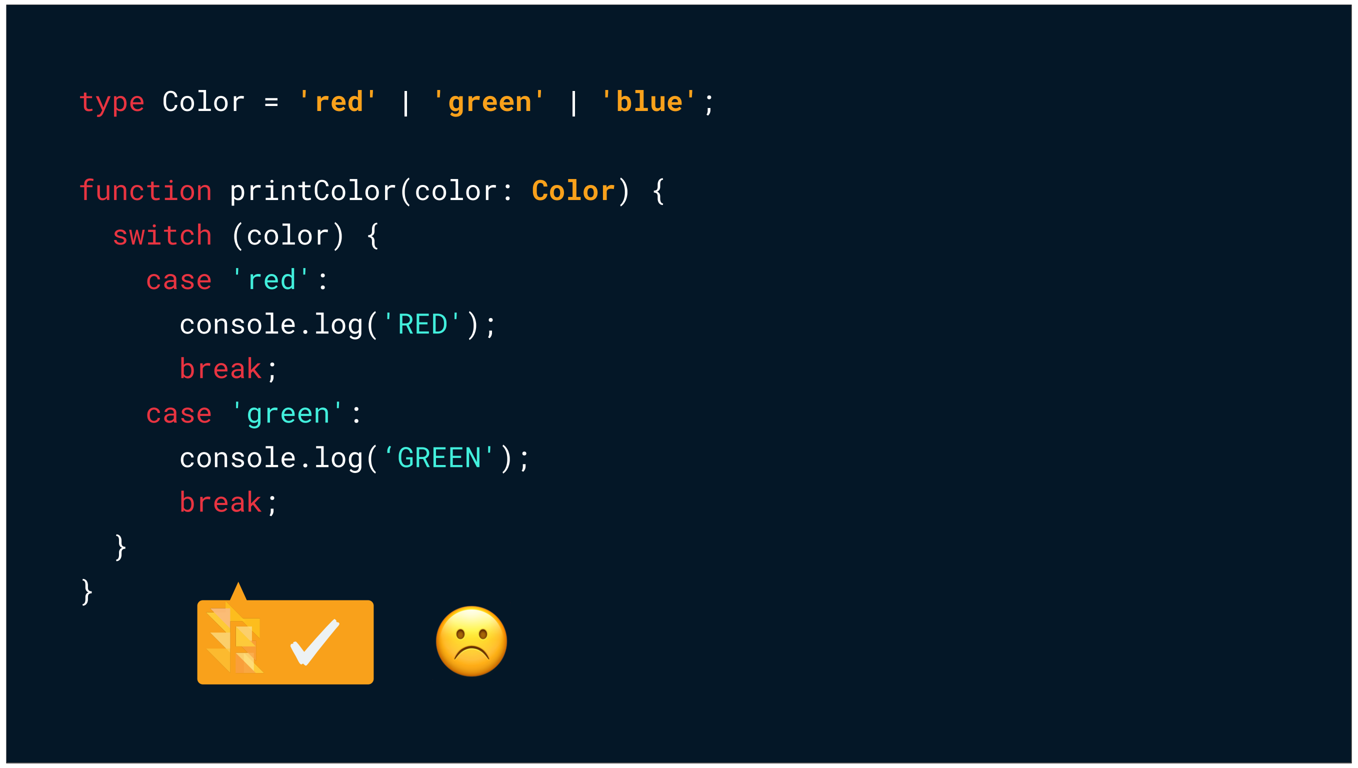


```
enum Color {
  Red,
  Green,
  Blue,
  Crimson
}

let model = match color {
  Color::Red => println!(“red"),
  Color::Green | Color::Blue => println!(“not red”)
};
```

! Color::Crimson not covered

But if we make sure that we exhaustively list the possibilities

Now our Type System can be helpful again!



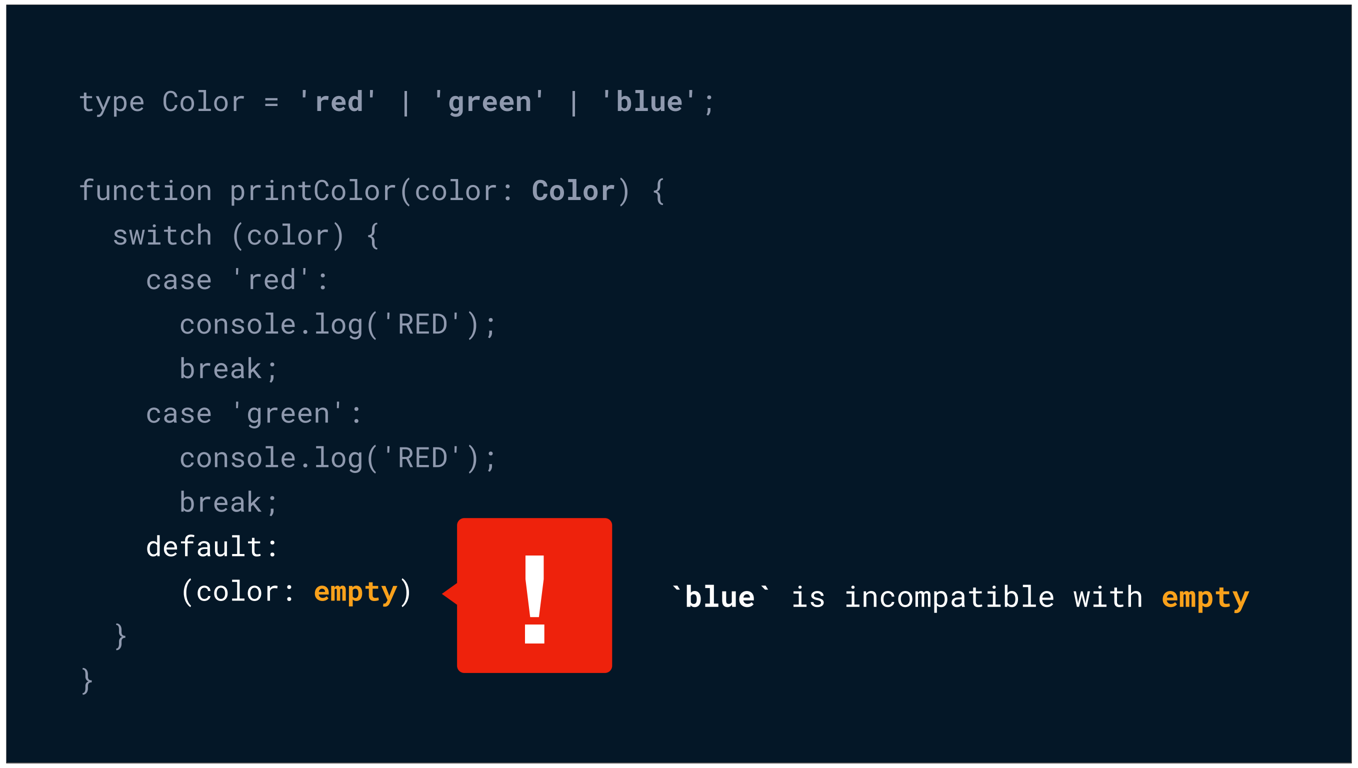
```
type Color = 'red' | 'green' | 'blue';

function printColor(color: Color) {
  switch (color) {
    case 'red':
      console.log('RED');
      break;
    case 'green':
      console.log('GREEN');
      break;
  }
}
```

✓ ☹️

So let's try the same thing in javascript

The type system is happy, even though we didn't cover every option

It turns out that neither Flow or TS support exhaustive matching (yet)



```
type Color = 'red' | 'green' | 'blue';

function printColor(color: Color) {
  switch (color) {
    case 'red':
      console.log('RED');
      break;
    case 'green':
      console.log('RED');
      break;
    default:
      (color: empty)
  }
}
```

`blue` is incompatible with empty

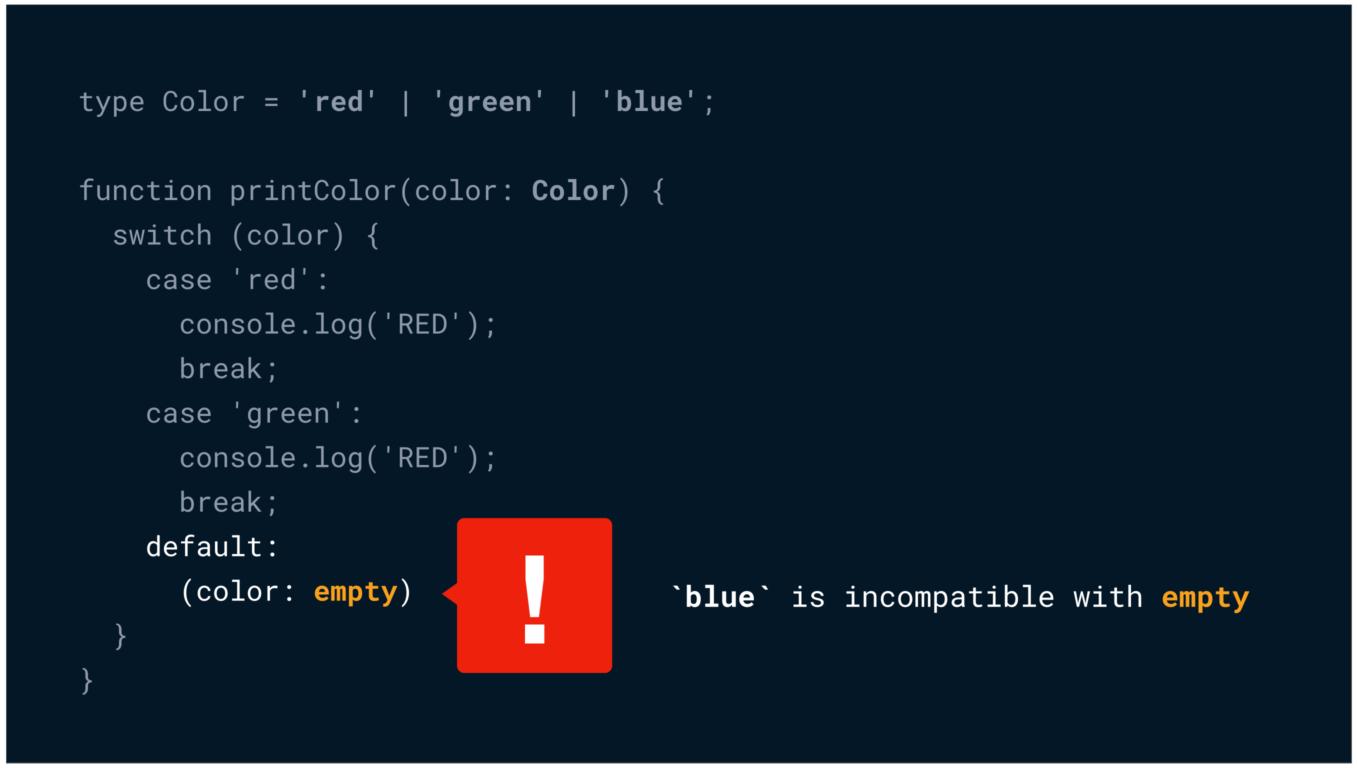
Flow has a special type called `empty`
(in TS it's `never`)

No types satisfy empty

So we can add it as a guard in a default case

We're asserting that this is type empty

But the type system can see that "blue" is going to hit this path, and will raise an error



```
type Color = 'red' | 'green' | 'blue';

function printColor(color: Color) {
  switch (color) {
    case 'red':
      console.log('RED');
      break;
    case 'green':
      console.log('RED');
      break;
    default:
      (color: empty)
  }
}
```

`blue` is incompatible with empty

So while we have to opt-in to this kind of checking
It’s SUPER useful

If we’ve structured our code this way
Let’s see how this might play out with our candy store

So let’s go back to our candy store

You decide to offer machine rentals.
Cotton candy machines are always a hit at children’s parties

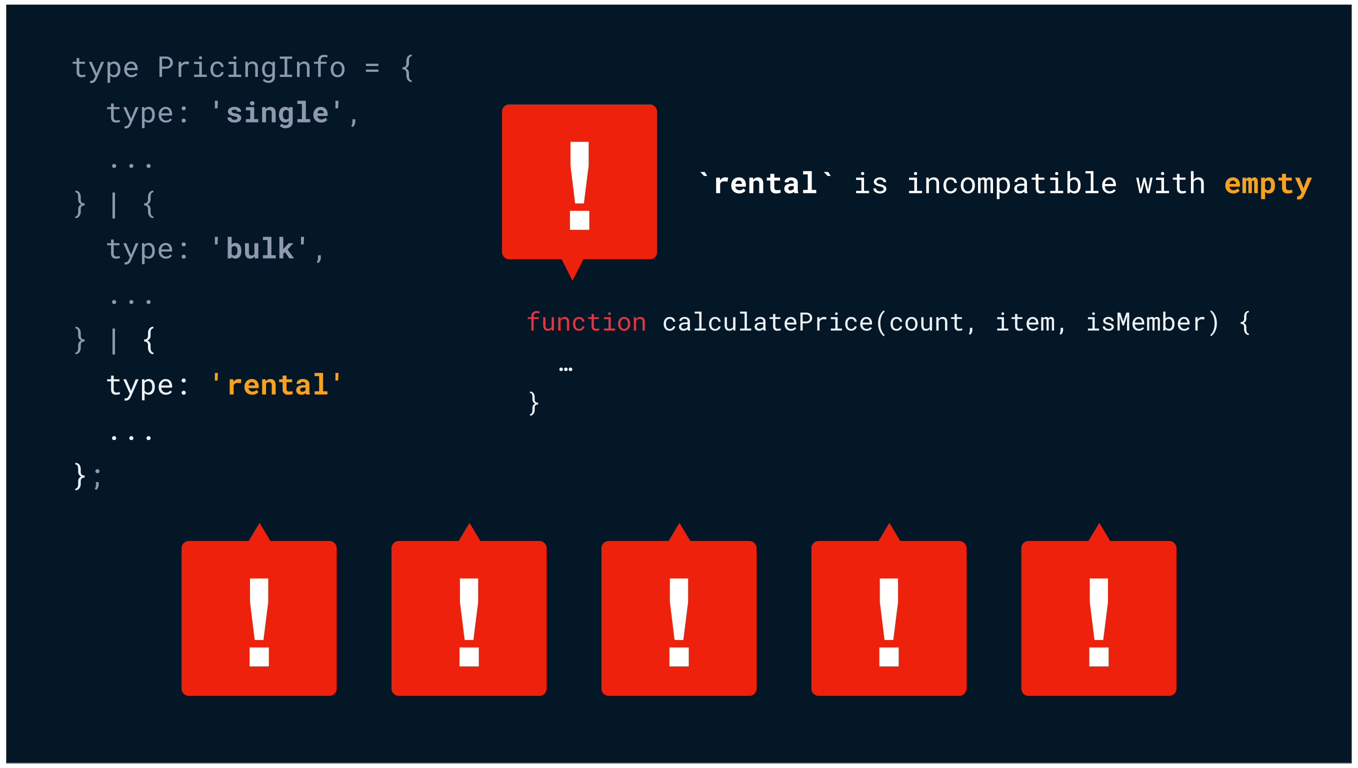
```
type PricingInfo = {
  type: 'single',
  ...
} | {
  type: 'bulk',
  ...
};
```

Now we have a logical place to extend to add the new data

```
type PricingInfo = {
  type: 'single',
  ...
} | {
  type: 'bulk',
  ...
} | {
  type: 'rental'
  ...
};
```

So we add a new rental type and the data that it needs

And because we've coded for exhaustion
we get errors



```
type PricingInfo = {
  type: 'single',
  ...
} | {
  type: 'bulk',
  ...
} | {
  type: 'rental'
  ...
};
```

`rental` is incompatible with empty

```
function calculatePrice(count, item, isMember) {
  …
}
```

We get an error in the function we were working on

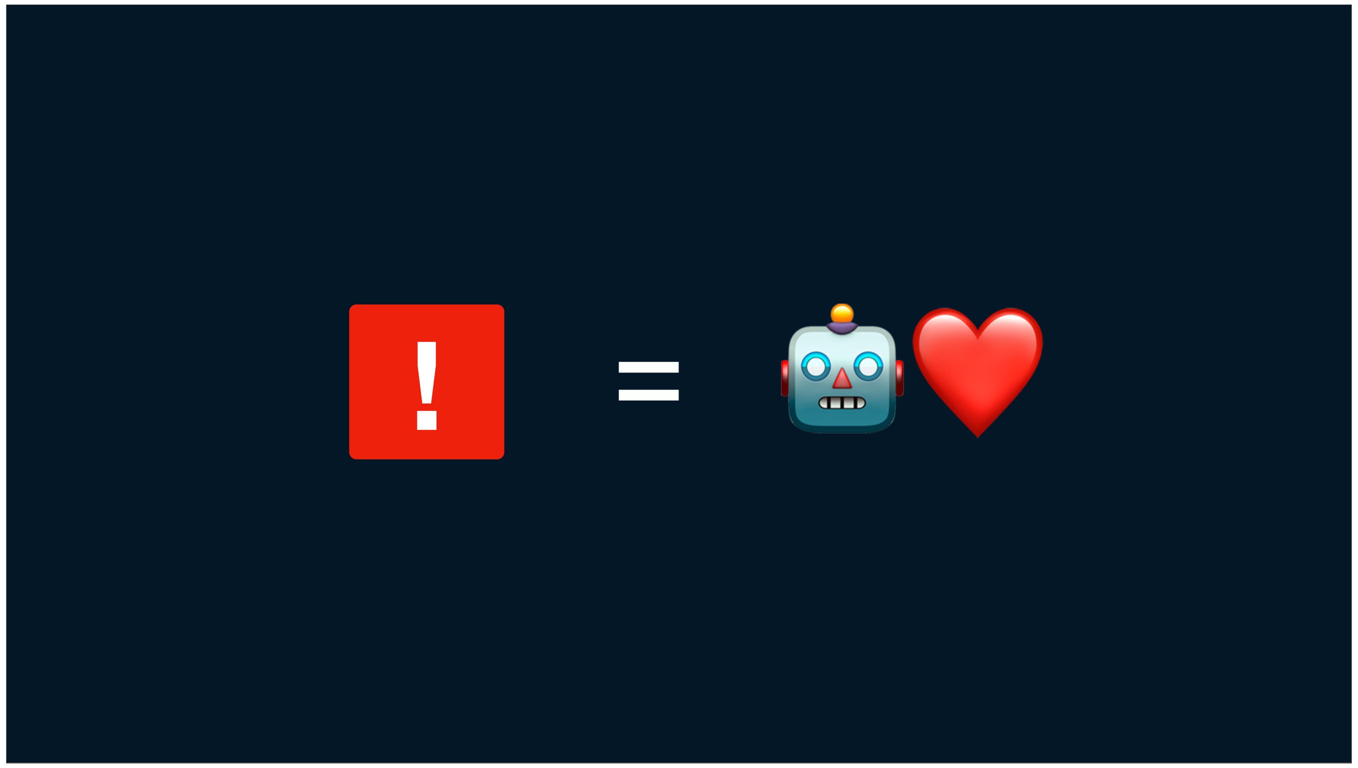
But also all … across… our application.

Anywhere we're accessing the data stored in this object

The compiler is raising an error, but it's really pointing to all the places we need to change

We don't have to have the whole system in our head

It's being helpful!

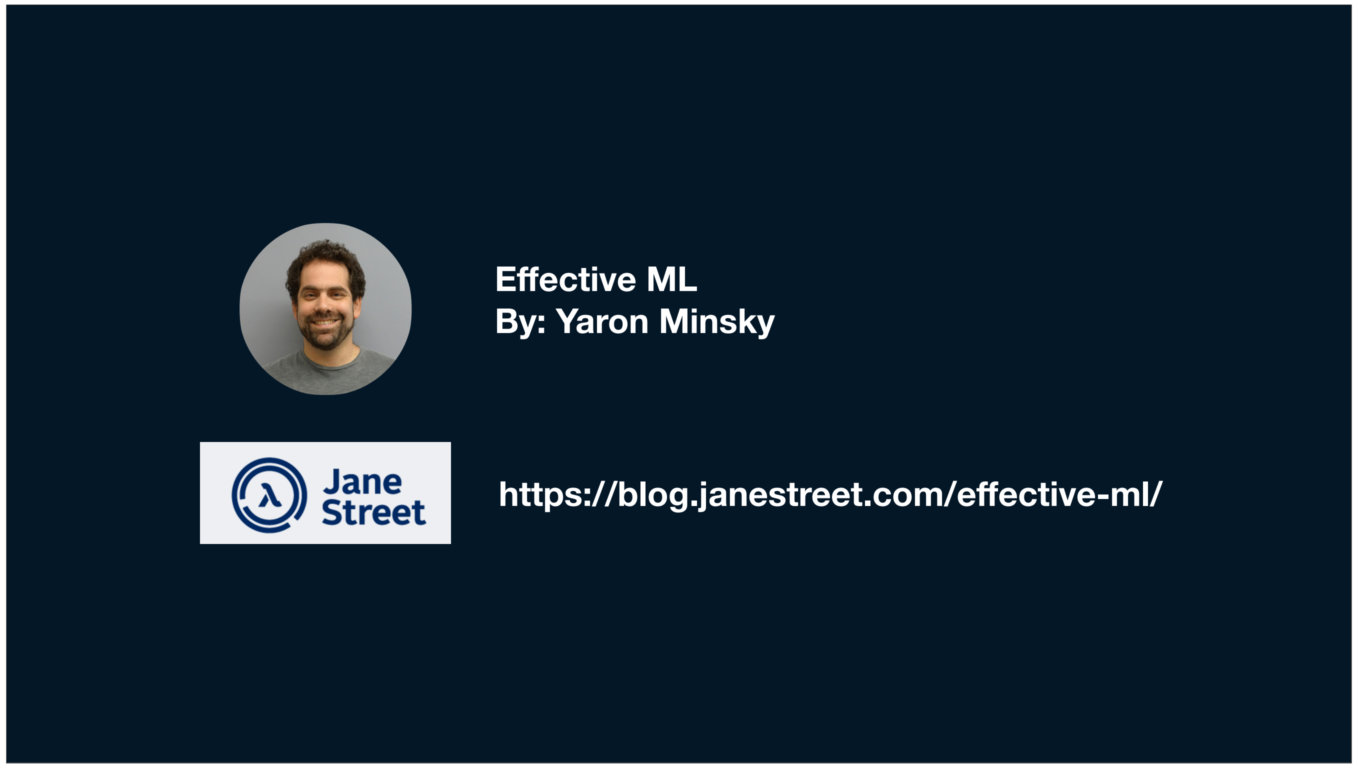


Errors are just your compiler's way of saying I love you and want you to be happy

Now this was something new to me

I was used to a compiler or type system as the final step
After I'd done all the work of building the feature
I'd have to fight with the type system for hours until it worked

By changing the way I wrote code slightly, the compiler became my friend
or at least my frenemy



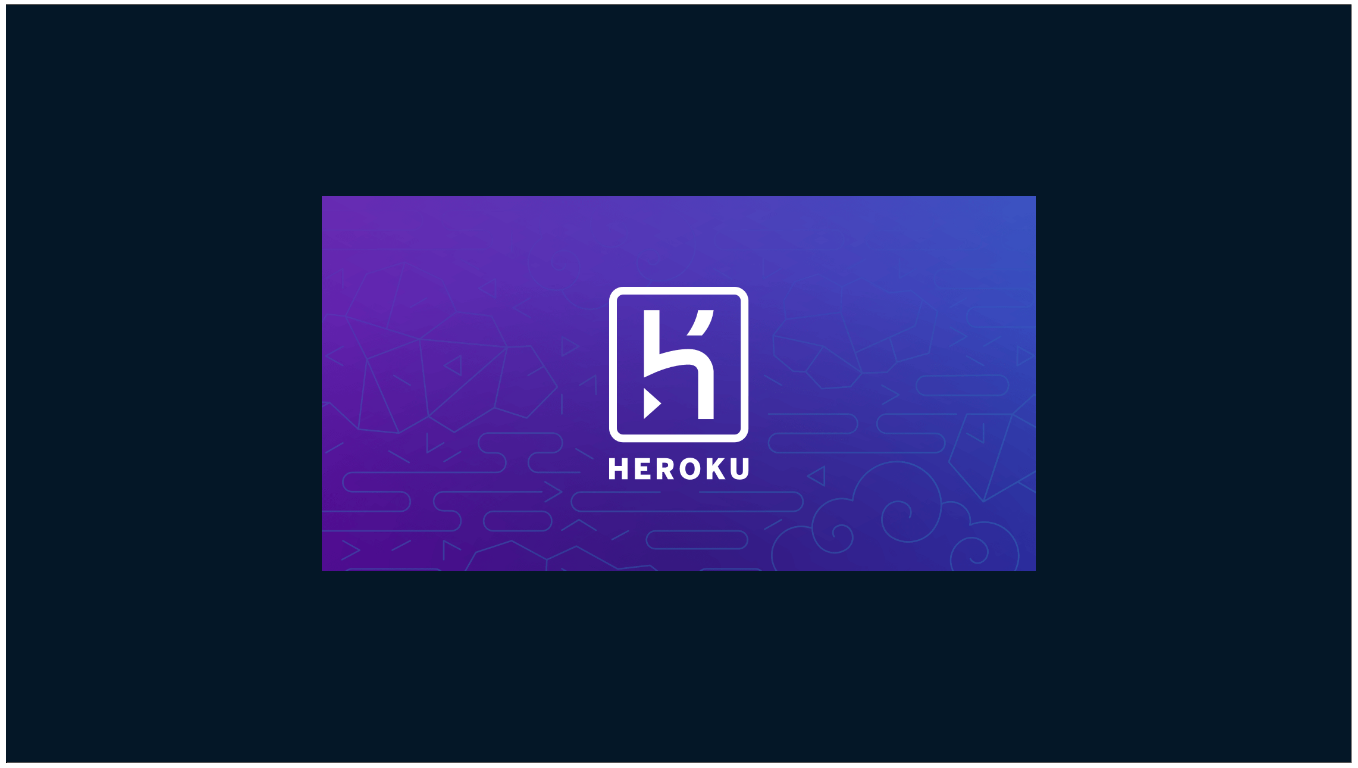
Effective ML
By: Yaron Minsky

https://blog.janestreet.com/effective-ml/

If this was interesting to you

I highly recommend watching this talk

You don’t really have to know OCaml to follow it
But the ideas are interesting
I stole some of them for this talk, but there are more I couldn’t include



Thanks!

[redacted]