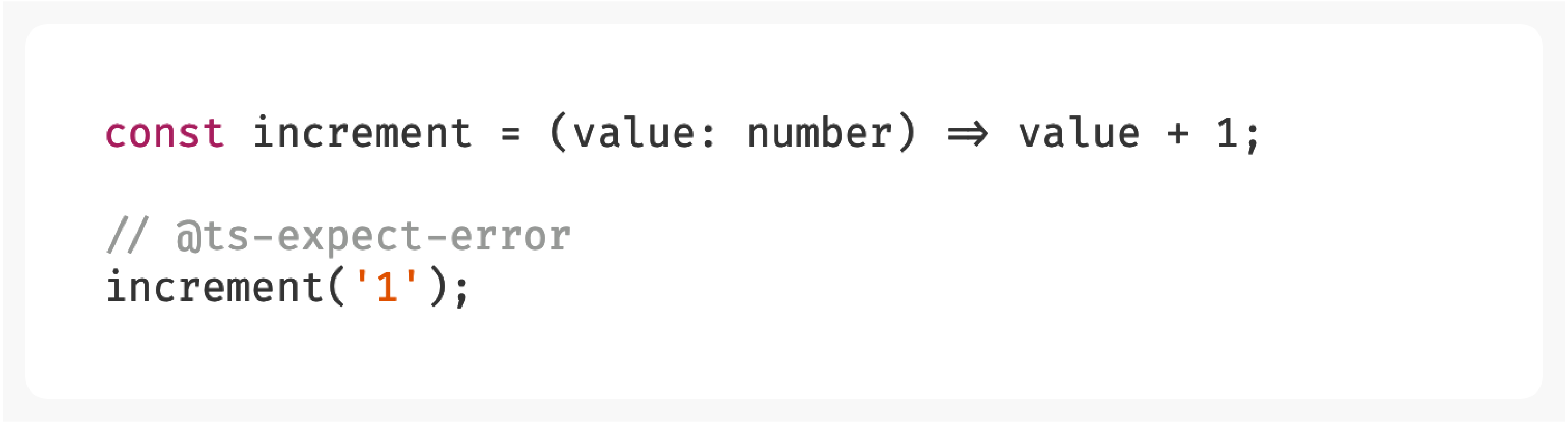
Opaque Types

TypeScript: a static type checker

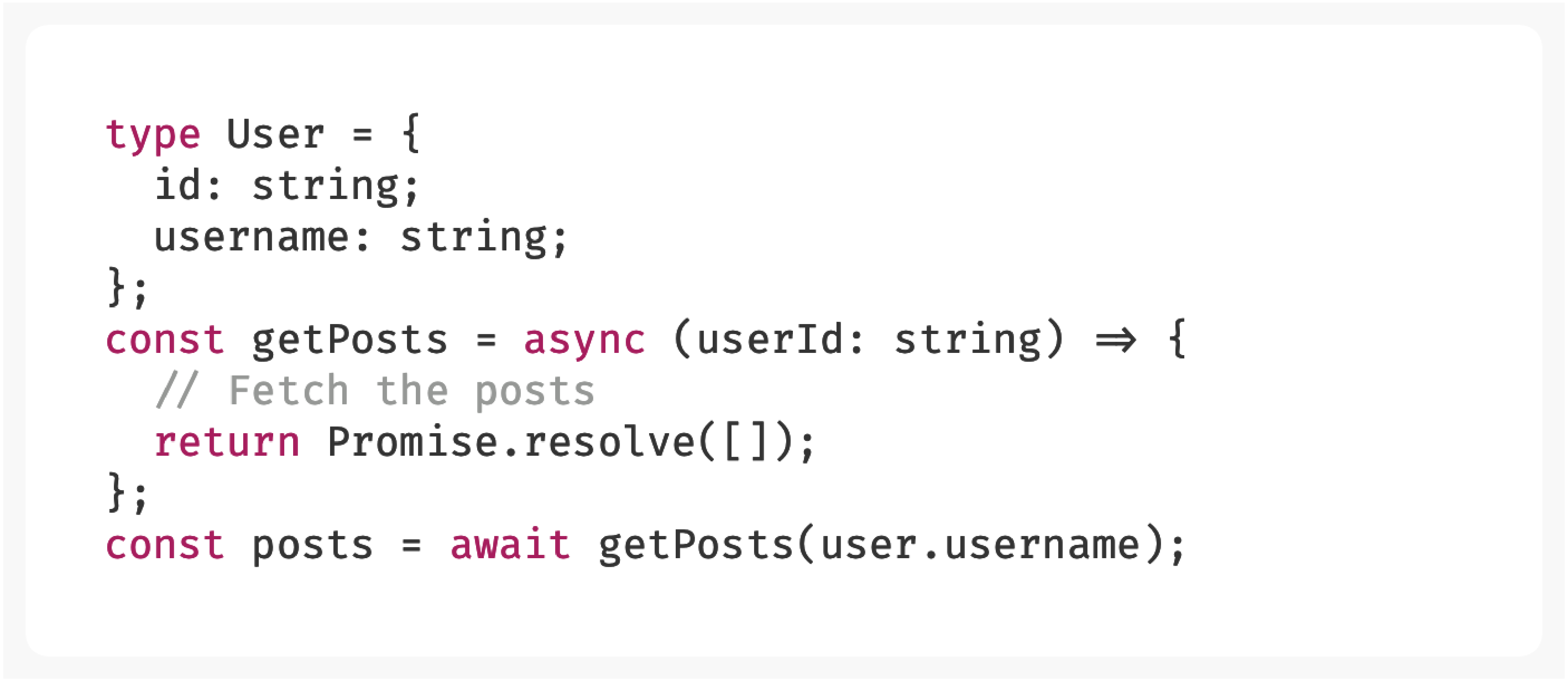
- Check the types at compile-time
- Prevent data types errors



```
const increment = (value: number) ⇒ value + 1;

// @ts-expect-error
increment('1');
```

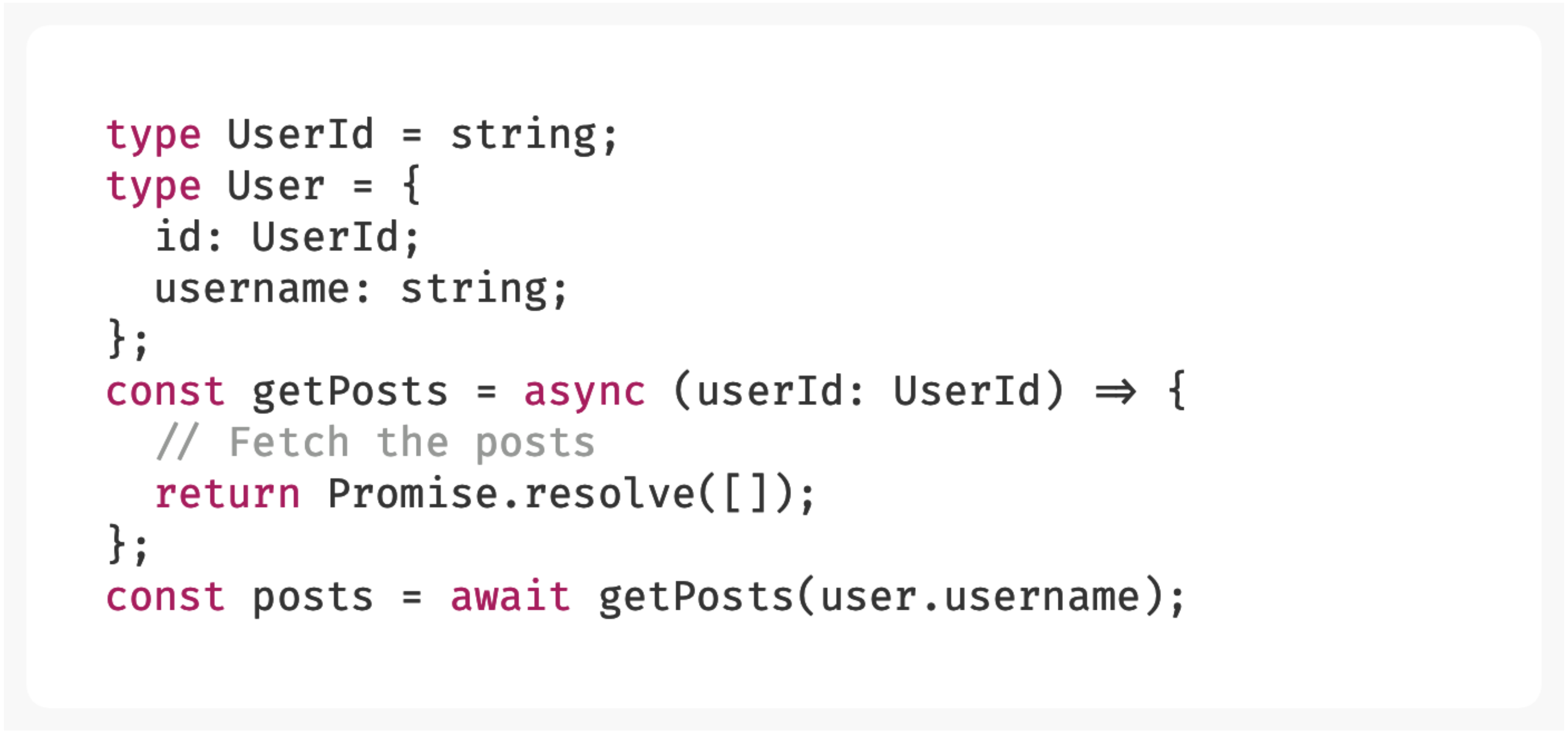
What’s wrong with TypeScript types?

TypeScript types are not enough 1/2



```typescript
type User = {
  id: string;
  username: string;
};
const getPosts = async (userId: string) => {
  // Fetch the posts
  return Promise.resolve([]);
};
const posts = await getPosts(user.username);
```

TypeScript types are not enough 2/2



```
type UserId = string;
type User = {
  id: UserId;
  username: string;
};
const getPosts = async (userId: UserId) ⇒ {
  // Fetch the posts
  return Promise.resolve([]);
};
const posts = await getPosts(user.username);
```

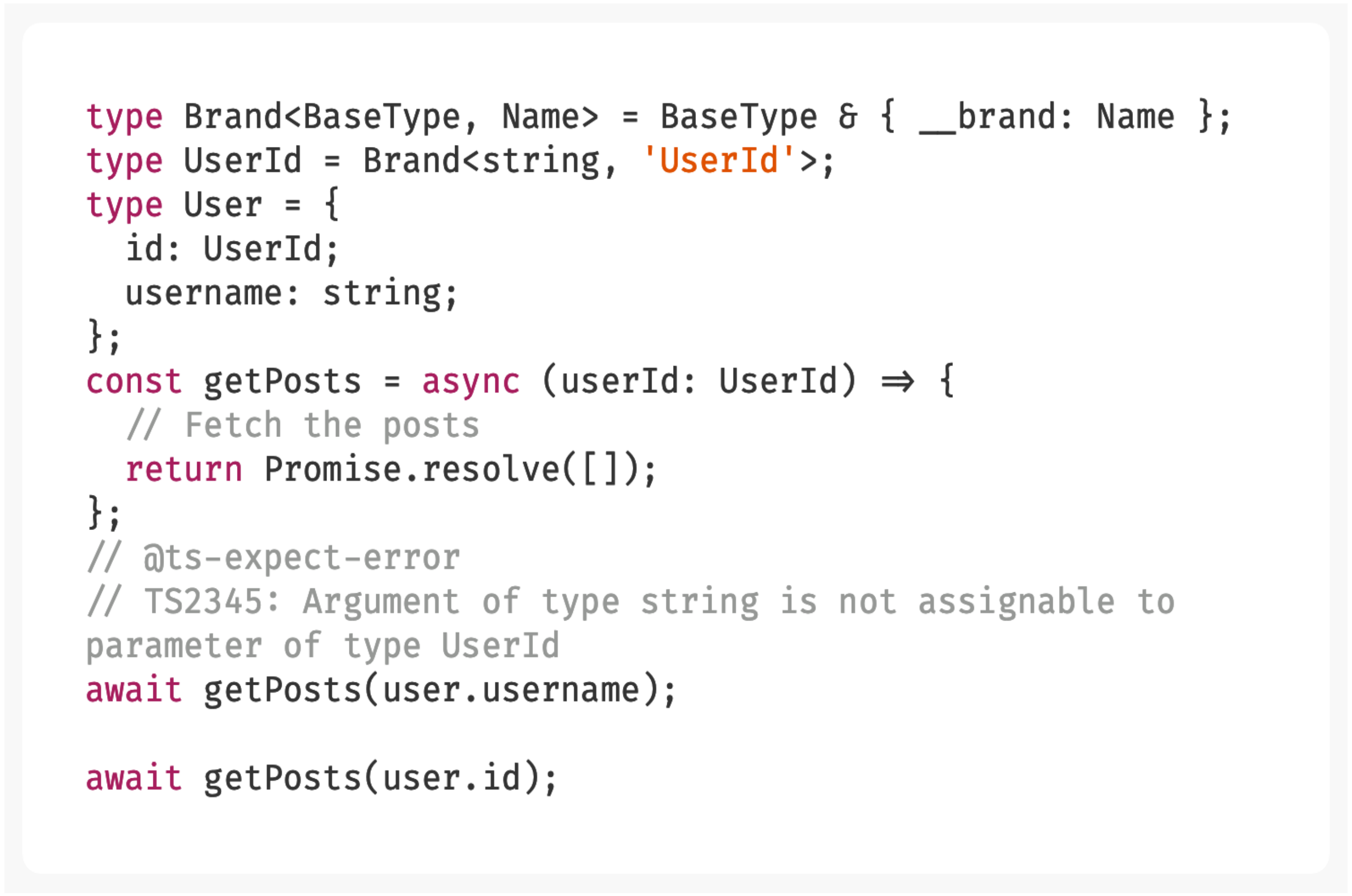
Opaque types to the rescue

Defining a new type

- Native feature in Flow but does not exist in TypeScript
- Accomplished by adding a tag to an existing type to create a new, more specific type

```
type Brand<BaseType, Name> = BaseType & { __brand: Name };
type UserId = Brand<string, 'UserId'>;
```

Putting it all together



```typescript
type Brand<BaseType, Name> = BaseType & { __brand: Name };
type UserId = Brand<string, 'UserId'>;
type User = {
  id: UserId;
  username: string;
};
const getPosts = async (userId: UserId) => {
  // Fetch the posts
  return Promise.resolve([]);
};
// @ts-expect-error
// TS2345: Argument of type string is not assignable to
parameter of type UserId
await getPosts(user.username);

await getPosts(user.id);
```

Why using opaque types?

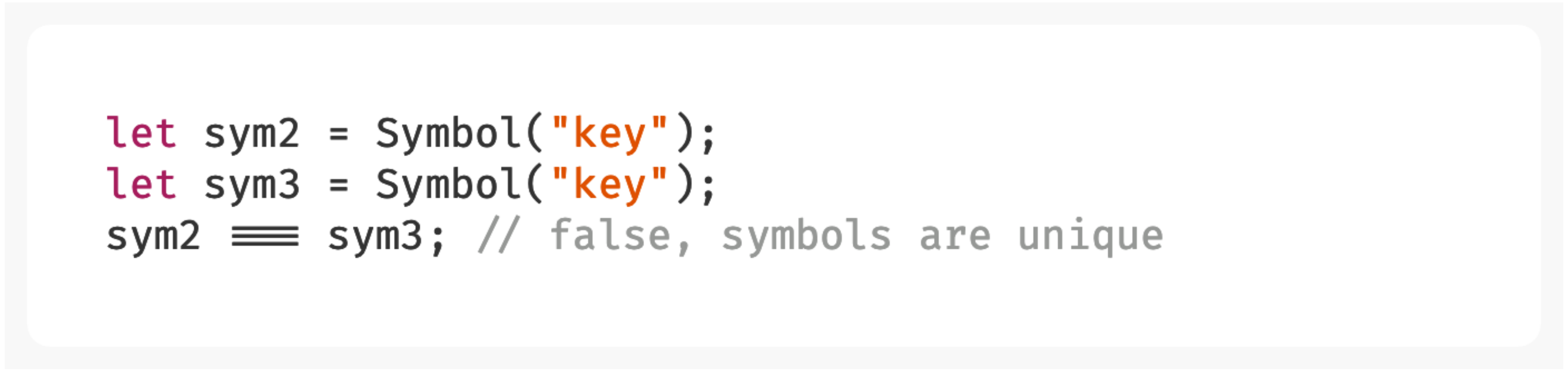
- Clarity
  - Explicit the domain type
- Safety and correctness
  - Add validation on the types
  - Allow the compiler to catch new errors
- Maintainability
  - Reduce ambiguity
  - Simplify refactoring by providing a clear distinction in the different types

Improving our new type

- Use a computed property key instead of a hardcoded name
- Use a `unique symbol` to prevent duplication
- Extract the Brand utility into its own file to prevent access to `__brand`

What are Symbols?

- Since ES6 `symbol` is a primitive type like `string` or `number`
- Created using the `Symbol()` function (cannot use `new Symbol()`)
- Symbols are immutable, and unique



```
let sym2 = Symbol("key");
let sym3 = Symbol("key");
sym2 === sym3; // false, symbols are unique
```

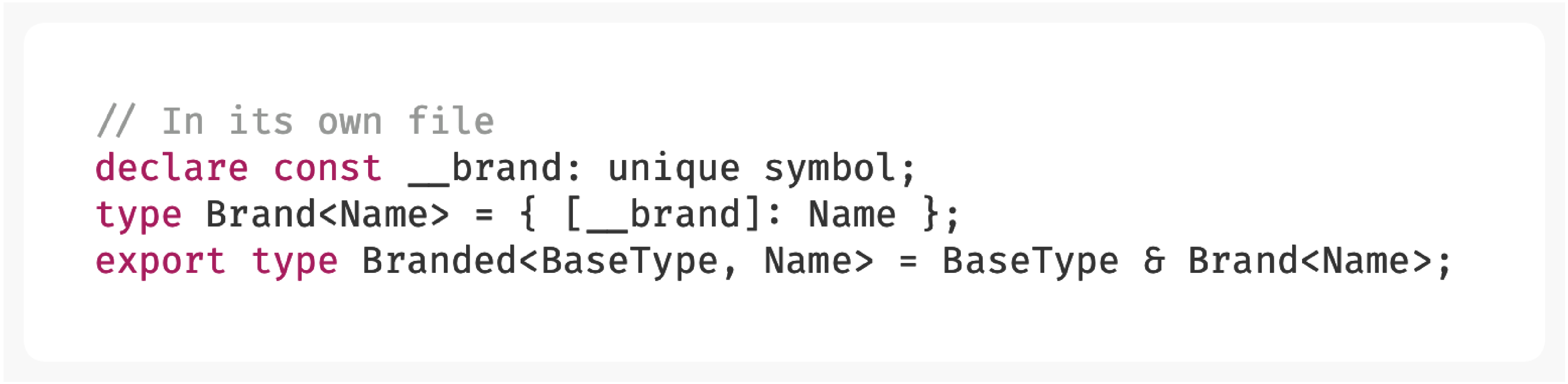
Symbols in real life

- Intended to be private and unique but ended up not being private
- Can be used to declare well-typed properties on objects
- You can see symbols using `Object.getOwnPropertySymbols()`
- But they are not visible using:
  - `for` loops
  - `Object.keys()`
  - `Object.getOwnPropertyNames()`
  - When converting an object to a JSON string

`unique symbol`

- Create a symbol in TypeScript without creating it in JavaScript: it won’t exist after compiling!
- Allowed only on `const` declarations and `readonly static` properties

Our final type



```
// In its own file
declare const __brand: unique symbol;
type Brand<Name> = { [__brand]: Name };
export type Branded<BaseType, Name> = BaseType & Brand<Name>;
```

Usage

Basic usage (not recommended)

- Cast your type as the opaque type

Add domain validation

- Help with the creation of function that can validate the format of the data
- Examples:
  - Validate an email
  - Ensure a number is between positive
  - Ensure an ID is for a specific object

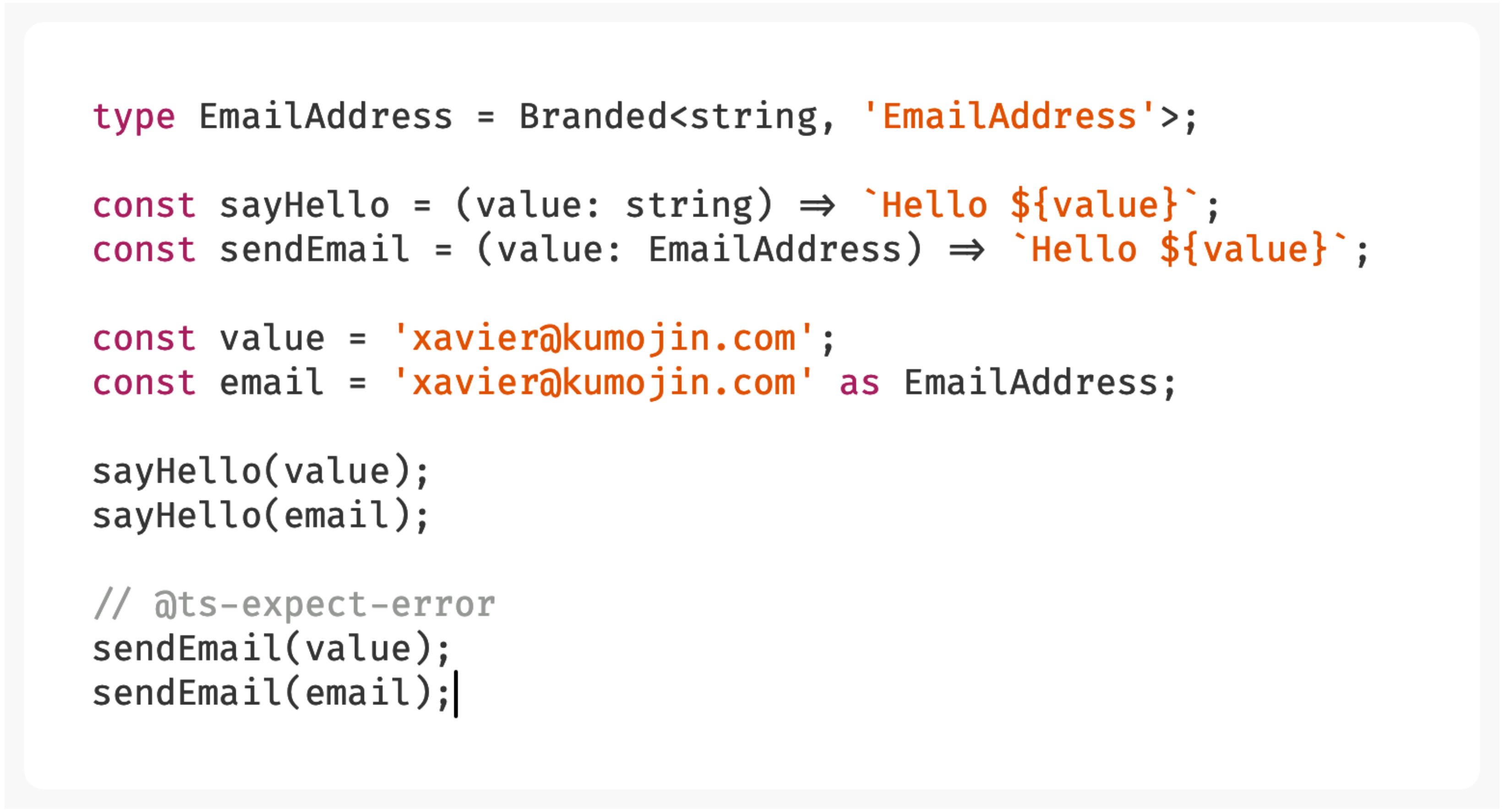
Using zod / valibot

- Use zod / valibot to parse your objects / API response and improve their types

Advanced opaque types

Weak Opaque Type

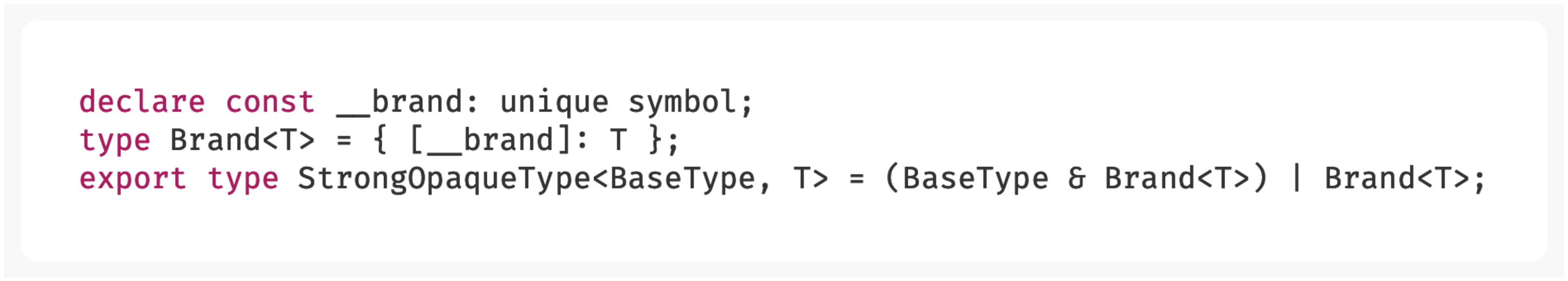
- Can be used either as the opaque type or the base type



```
type EmailAddress = Branded<string, 'EmailAddress'>;

const sayHello = (value: string) => `Hello ${value}`;
const sendEmail = (value: EmailAddress) => `Hello ${value}`;

const value = 'xavier@kumojin.com';
const email = 'xavier@kumojin.com' as EmailAddress;

sayHello(value);
sayHello(email);

// @ts-expect-error
sendEmail(value);
sendEmail(email);
```

Strong Opaque Type

- Can be used either as the opaque type or the base type but needs to be explicitly casted to the base type
- Useful if we don't accidentally want to use the more specific type for more general cases without a clear exception being made via explicit casting



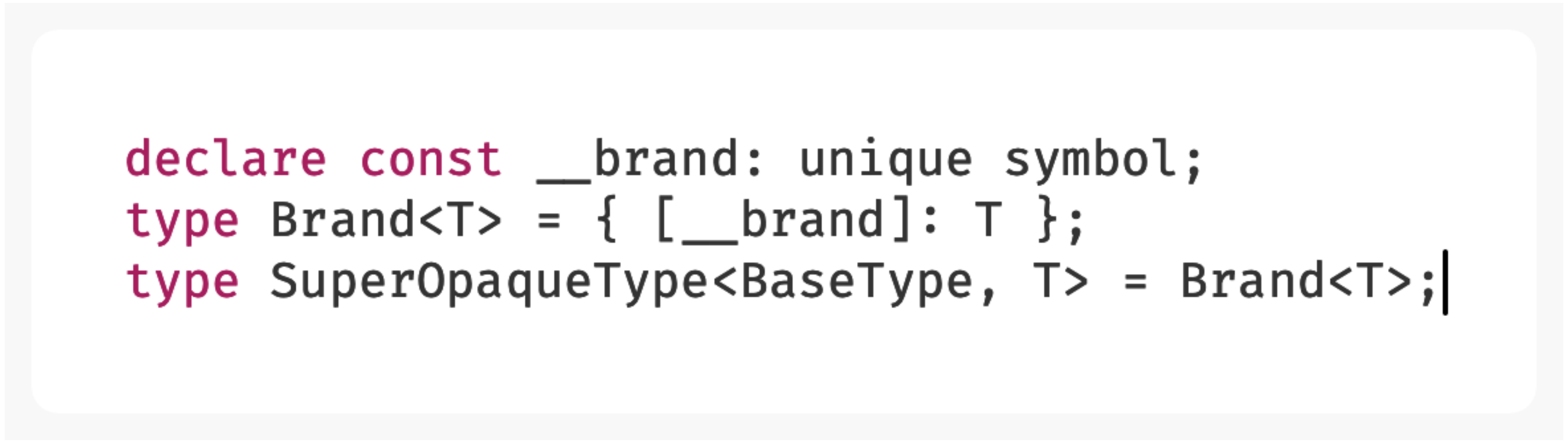
```
declare const __brand: unique symbol;
type Brand<T> = { [__brand]: T };
export type StrongOpaqueType<BaseType, T> = (BaseType & Brand<T>) | Brand<T>;
```

Strong Opaque Type - Example

```typescript
type EmailAddress = StrongOpaqueType<string, 'EmailAddress'>;

const sayHello = (value: string) => `Hello ${value}`;
const sendEmail = (value: EmailAddress) => `Hello ${value}`;

const value = 'xavier@kumojin.com';
const email = 'xavier@kumojin.com' as EmailAddress;

sayHello(value);
sayHello(email as string);

// @ts-expect-error
sendEmail(value);
sendEmail(email);
```

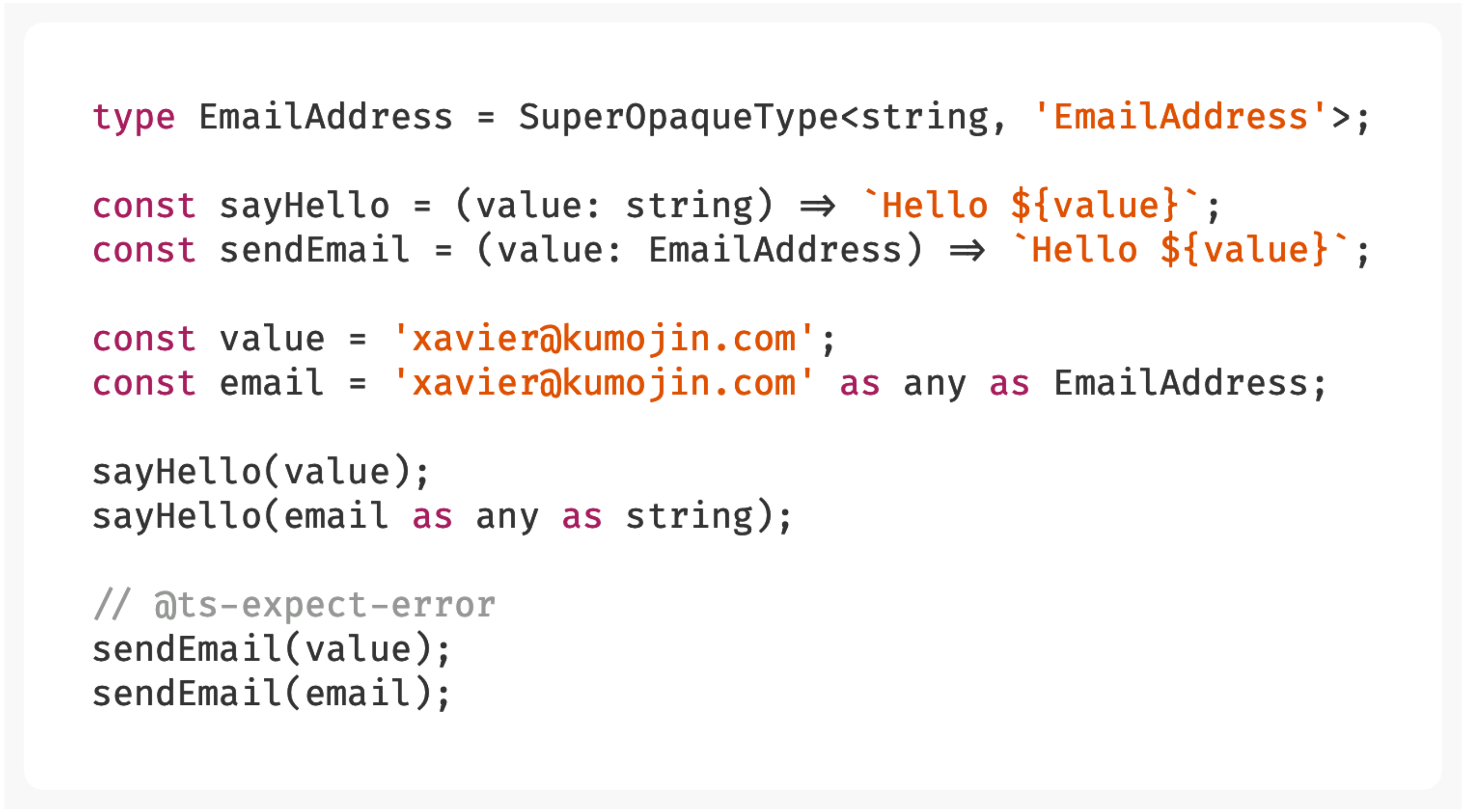
Super Opaque Type

- Can not be explicitly or implicitly casted to their base type
- This is useful for cases where we are truly making a new type that is unrelated any base type
- This is like comparing number to string



```
declare const __brand: unique symbol;
type Brand<T> = { [__brand]: T };
type SuperOpaqueType<BaseType, T> = Brand<T>;
```

Super Opaque Type - Example



```typescript
type EmailAddress = SuperOpaqueType<string, 'EmailAddress'>;

const sayHello = (value: string) => `Hello ${value}`;
const sendEmail = (value: EmailAddress) => `Hello ${value}`;

const value = 'xavier@kumojin.com';
const email = 'xavier@kumojin.com' as any as EmailAddress;

sayHello(value);
sayHello(email as any as string);

// @ts-expect-error
sendEmail(value);
sendEmail(email);
```

Question?