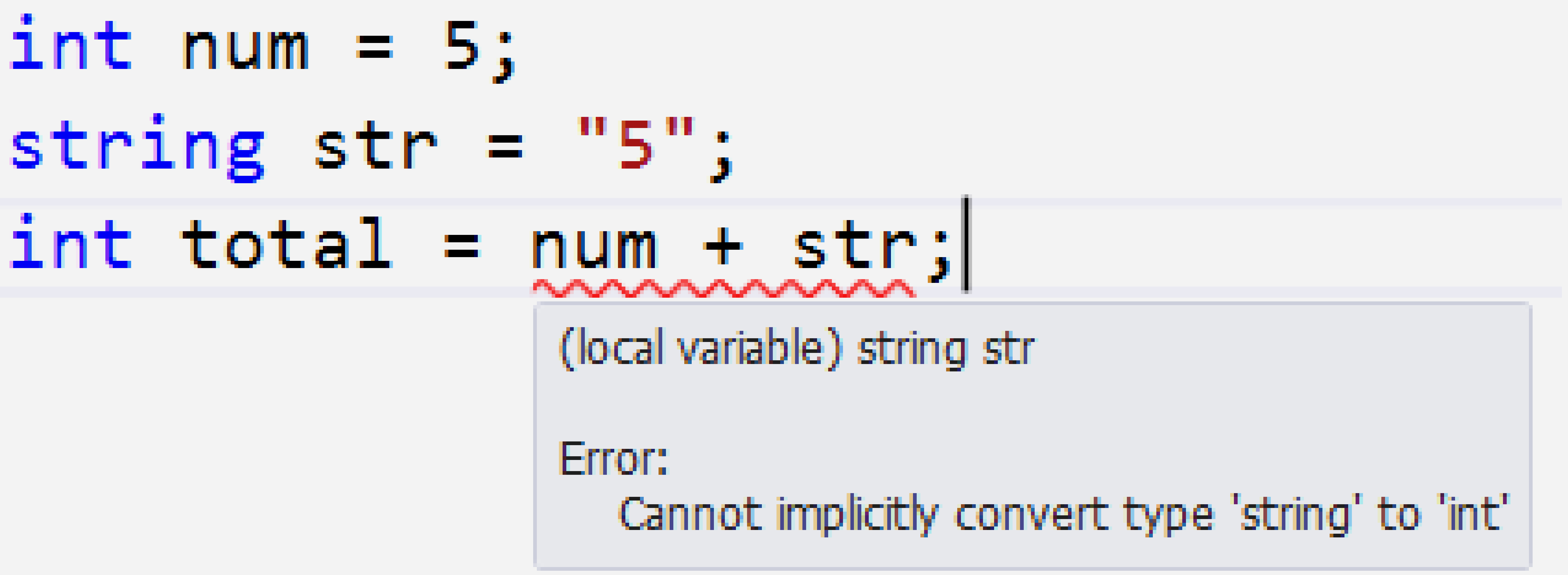
This talk is about **types.**

This talk is about

types.



```
int num = 5;
string str = "5";
int total = num + str;
```

```
(local variable) string str

Error:
    Cannot implicitly convert type 'string' to 'int'
```

This talk is about

using types.

This talk is about

using types in

gleam

👩‍💻 Frontend Developer

at **data2impact**

👩‍💻 **Frontend Developer**

at **data2impact**

💖 **Developer Relations**

at **xyflow**

👩‍💻 **Frontend Developer**
at **data2impact**

💖 **Developer Relations**
at **react flow**

🌪️ **Too Many Things**
at **gleam**

Phantom types and the builder pattern

Phantom types? and the builder pattern

A detour on **generics**

```
pub type List {
  Head(val: Int, rest: List)
  Tail
}
```

```
pub type List {
  Head(val: Int, rest: List)
  Tail
}
```

```
pub type List(a) {
  Head(val: a, rest: List(a))
  Tail
}
```

Phantom types?and the builder pattern

```
pub type List(a) {
  Head(val: a, rest: List(a))
  Tail
}
```

```
pub type List(a) {
  Head(val: a, rest: List(a))
  Tail
}
```

Phantom types **don’t exist at runtime** 👻

```
pub type List(a) {
  Head(val: a, rest: List(a))
  Tail
}

fn example() {
  let x: List(Int) = Tail
  let y: List(String) = Tail

  x == y
}
```

```
error: Type mismatch
  ┌─  ...
  │
9 │    x == y
  │         ^
  │

Expected type: List(Int)
Found type: List(String)
```

So what are they good for?

So what are they good for?

- Ids

```
pub opaque type Id {
  Id(Int)
}

pub fn from_int(id: Int) -> Id {
  Id(id)
}
```

```
pub fn example() {
  let post_id = from_int(1)
  let user_id = from_int(2)

  upvote(user_id, post_id)
}
```

```
pub fn example() {
  let post_id = from_int(1)
  let user_id = from_int(2)

  upvote(user_id, post_id)
}

pub fn upvote(
  post_id: Id,
  user_id: Id
) -> Nil { ... }
```

```
pub opaque type Id(kind) {
  Id(Int)
}

pub fn from_int(id: Int) -> Id(kind) {
  Id(id)
}
```

```
pub type User { ... }

pub type Post { ... }

pub fn upvote(
  post: Id(Post),
  user: Id(User),
) { ... }
```

```
pub fn example() {
  let post_id: Id(Post) = from_int(1)
  let user_id: Id(User) = from_int(2)

  upvote(user_id, post_id)
}
```

```
error: Type mismatch
   ┌─ ...
   │
17 │   upvote(user_id, post_id)
   │          ^^^^^^^
   │

Expected type: Id(Post)
Found type: Id(User)
```

```
pub opaque type Id(kind) {
  Id(val: Int)
}

pub fn new() -> Id(kind) {
  Id(val: int.random(1234))
}

pub fn show(id: Id(kind)) -> String {
  int.to_string(id.val)
}
```

So what are they good for?

- Ids
- Validation

```
pub type Password = String

pub fn is_valid(pass: Password) -> Bool {
  ...
}

pub fn create_user(
  username: String,
  password: Password
) -> Result(User, Error) {
  ...
}
```

```
pub opaque type Password(validation) {
  Password(String)
}

pub type Invalid

pub type Valid

pub fn from_string(str) -> Password(Invalid) {
  Password(str)
}
```

```
pub fn validate(
  password: Password(Invalid)
) -> Result(Password(Valid), Reason)

pub fn suggest(
  password: Password(Invalid)
) -> String

pub fn create_user(
  username: String,
  password: Password(Valid)
) -> User
```

```
pub fn validate(
  password: Password(Invalid)
) -> Result(Password(Valid), Reason)

pub fn suggest(
  password: Password(Invalid)
) -> String

pub fn create_user(
  username: String,
  password: Password(Valid)
) -> User
```

```
pub fn validate(
  password: Password(Invalid)
) -> Result(Password(Valid), Reason)

pub fn suggest(
  password: Password(Invalid)
) -> String

pub fn create_user(
  username: String,
  password: Password(Valid)
) -> User
```

Phantom types **restrict APIs** so you can focus on the happy path.

Phantom types and the builder pattern

Phantom types and the **builder pattern**

```
pub opaque type ButtonConfig {
  ButtonConfig(
    label: String,
    icon : Option(Icon),
    colour: Option(Colour),
    ...
  )
}

pub new(label: String) -> ButtonConfig {
  ButtonConfig(label, None, None)
}
```

```
pub opaque type ButtonConfig {
  ButtonConfig(
    label: String,
    icon : Option(Icon),
    colour: Option(Colour),
    ...
  )
}

pub new(label: String) -> ButtonConfig {
  ButtonConfig(label, None, None, ...)
}
```

```
pub fn with_colour(config, colour) {
  Config(..config, colour: Some(colour))
}

pub fn with_icon(config, icon) {
  Config(..config, icon: Some(icon))
}

...
```

```
new(“wibble”)
|> with_icon(Sparkles)

new(“wobble”)
|> with_colour(Error)
|> is_disabled(True)

new(“woo”)
|> with_icon(Confetti)
|> with_style(Outline)
|> with_colour(Success)
|> with_icon(Sparkles)
```

```
new("wibble")
|> with_icon(Sparkles)

new("wobble")
|> with_colour(Error)
|> is_disabled(True)

new("woo")
|> with_icon(Confetti)
|> with_style(Outline)
|> with_colour(Success)
|> with_icon(Sparkles)
```

```
new(“wibble”)
|> with_icon(Sparkles)

new(“wobble”)
|> with_colour(Error)
|> is_disabled(True)

new(“woo”)
|> with_icon(Confetti)
|> with_style(Outline)
|> with_colour(Success)
|> with_icon(Sparkles)
```

```
new(“wibble”)
|> with_icon(Sparkles)

new(“wobble”)
|> with_colour(Error)
|> is_disabled(True)

new(“woo”)
|> with_icon(Confetti)
|> with_style(Outline)
|> with_colour(Success)
|> with_icon(Sparkles)
```

👻 boo.

```
pub opaque type ButtonConfig(has_icon) {
  ButtonConfig(
    label: String,
    icon : Option(Icon),
    ...
  )
}

pub type NoIcon

pub type HasIcon
pub fn new(...) -> ButtonConfig(NoIcon) {
  ...
}

pub fn with_icon(
  config: ButtonConfig(NoIcon),
  icon: Icon
) -> ButtonConfig(HasIcon) {
  ...
}
```

```
new("woo")
|> with_icon(Confetti)
|> with_style(Outline)
|> with_colour(Success)
|> with_icon(Sparkles)

error: Type mismatch

   ┌  ...
19 │    |> with_colour(Success)
          ^^^^^^^^^^^^^^^^^^^^

Expected type: ButtonConfig(NoIcon)
Found type: ButtonConfig(HasIcon)
```

```
new("woo")
|> with_icon(Confetti)
|> with_style(Outline)
|> with_colour(Success)
|> with_icon(Sparkles)
```

```
error: Type mismatch

   ┌   ...
19 │     |> with_colour(Success)
           ^^^^^^^^^^^^^^^^^^^^

Expected type: ButtonConfig(NoIcon)
Found type: ButtonConfig(HasIcon)
```

So what are they good for?

- Ids
- Validation
- The Real World ™

Phantom types **don’t exist at runtime**. We can use them to **restrict APIs** and focus on the **happy path**. 👻

Thanks for listening!