What Are They?

A brief introduction to functors, and how they'll tie into this course. (4 min. read)

value: 20

FP defines its own types, and functors are among the most basic. Think of it as a container that holds any value. It can be a string, date, boolean, array, object, etc.

This container **must have a `map` method** to be considered a functor.

Here's a basic example

```
const Identity = (x) => ({
  value: x,
  map: (fn) => Identity(fn(x))
});
```

This function, `Identity`, takes a value and returns a functor.

```
const name = Identity('Bobo');

console.log(name);
```

Logging `name` shows us a functor of this shape

```
{ value: 'Bobo', map: [Function: map] }
```

How would you change this functor's inner value, `'Bobo'`?

This is JavaScript, so we could mutate it.

```
const name = Identity('Bobo');

name.value = name.value.toUpperCase();

console.log(name);
```

But we know FP actively discourages this practice, so what if we used `map` instead?

```
const name = Identity('Bobo');
const newName = name
  .map(value => value.toUpperCase());

console.log({ name, newName });
```

Without affecting the previous functor, we've created a brand new functor with our desired value!

And since `map` must return the same type, we can chain it similar to array's `map`.

```
const name = Identity('Bobo');
const newName = name
  .map(value => value.slice(0, 3))
  .map(value => `My name is not ${value}!`)
  .map(value => value.toUpperCase())

console.log({ name, newName });
```

This way is immutable, easier to read, and composes better.

And when you need to extract it, just access its `.value` property.

Summary

- Functors are containers you can map over.
- Functors can hold any value
- `map` is a composable, immutable way to change a functor's value.

Who Cares?

Lenses use functors. We'll be diving into them *very* soon.