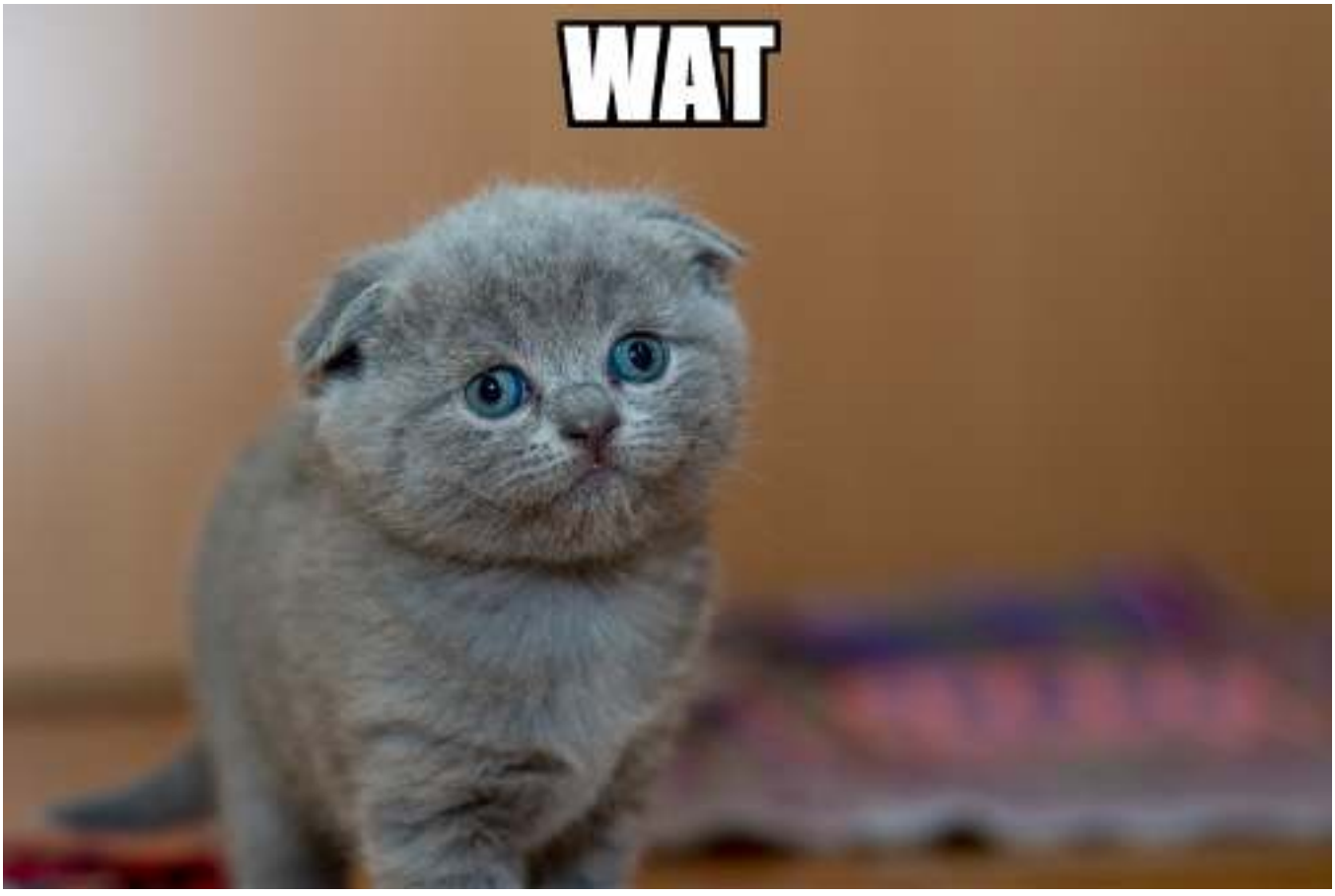
Lenses Under the Hood

We explore what makes lenses tick, and experiment with functors along the way. (12-15 min. read)

Check out Ramda's `lens` source code. As of this course, it looks like this

```
var lens = _curry2(function lens(getter, setter) {
  return function(toFunctorFn) {
    return function(target) {
      return map(
        function(focus) {
          return setter(focus, target);
        },
        toFunctorFn(getter(target))
      );
    };
  };
});
```



There's a lot to unpack here but I think we can do it. Starting with the first line...

Currying

Look at the top of the source code file.

```
import _curry2 from './internal/_curry2';
```

Ok, not too scary–they imported an internal `_curry2` function. Why?

```
var lens = _curry2(function lens(getter, setter) {
```

It's currying `lens`. Ramda curries everything so `_curry2` seems to be specialized for functions with two arguments.

Why a specialized curry()?

Ramda's `curry` is dynamic, and handles all types of functions with differing argument counts.

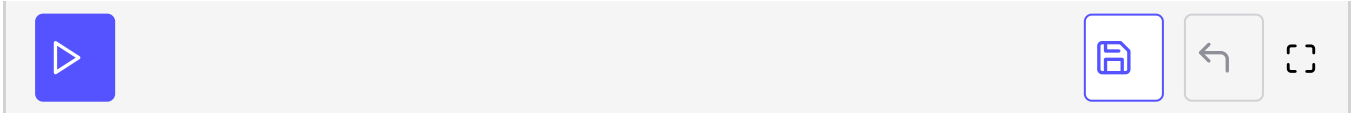
If you know how many arguments you need, however, using a specialized `curry` can optimize your function's run-time. You can ignore handling any other argument case, because you know your function expects 2, 3, 4, etc arguments.

Next line!

```
var lens = _curry2(function lens(getter, setter) {
  return function(toFunctorFn) {
```

So `lens` returns a function after receiving its getter and setter. Let's test that out.

```
import { assoc, lens, prop } from 'ramda';

const name = lens(prop('name'), assoc('name'));

console.log(name.toString());
```



Yep! This returned a function that takes one parameter, `toFunctorFn`. "To

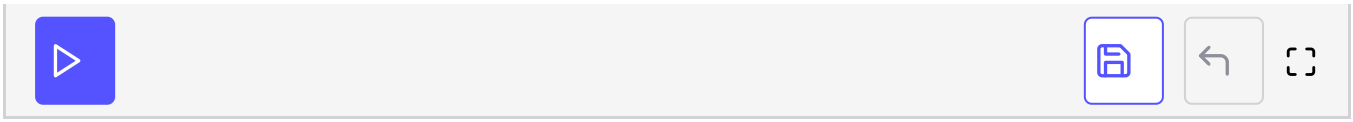
Functor Function”

Sounds like a function that turns something into a **functor**. We know from the previous section that functors are containers that can hold any value.

Using Functors

What’s the next step after giving `lens` a getter/setter? Pass it to `view`, `set`, or `over`.

```
import { assoc, lens, prop, view } from 'ramda';

const name = lens(prop('name'), assoc('name'));
const result = view(name, { name: 'Bobo' });

console.log({ result });
```



That’s pretty cool. With the getter/setter a lens looks like this

```
function (toFunctorFn) {
  return function (target) {
    return map(function (focus) {
      return setter(focus, target);
    }, toFunctorFn(getter(target)));
  };
}
```

After giving it to `view`, though...

```
view(name, { name: 'Bobo' });
```

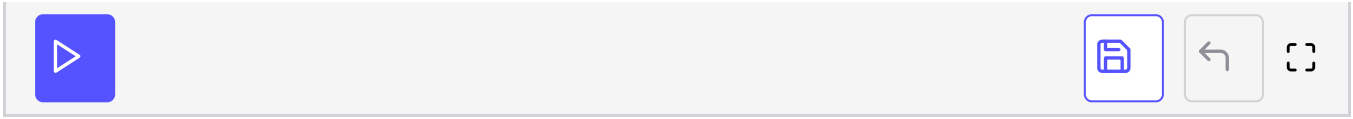
...everything’s magically resolved and we get `'Bobo'`.

View Magic

That means `view` satisfied all of `lens`’ requirements in a single shot, so it’s our next point of investigation. Take a look at its source code. There’s a variable, `Const`.

```
var Const = function(x) {
  return {
```

```
    value: x,
    'fantasy-land/map': function() { return this; }
  };
};

console.log(Const('Hello World'));
```



It creates a functor with two properties

1. `value`
2. `fantasy-land/map` method

We've already seen this. It wants to override `map`'s functionality.

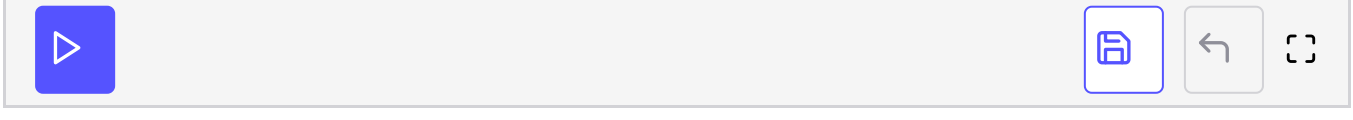
Now look at `view`'s implementation.

```
var view = _curry2(function view(lens, x) {
  return lens(Const)(x).value;
});
```

`Const` is the `toFunctorFn` here. It will tell `lens` how to turn something into a functor.

Let's extract that step into our own playground to experiment a little. We just need to define our own `Const` and give it to the `name` lens we created earlier.

```
import { assoc, lens, prop, view } from 'ramda';

// Define Const
var Const = function(x) {
  return {
    value: x,
    'fantasy-land/map': function() { return this; }
  };
};

const name = lens(prop('name'), assoc('name'));

// And use it here
const withFunctorFn = name(Const);

console.log(withFunctorFn.toString());
```



Yet Another Function

It returned another function, when will the madness end?!

```
function (target) {
  return map(function (focus) {
    return setter(focus, target);
  }, toFunctorFn(getter(target)));
}
```

This one needs a `target`. Let's see how `view` satisfies this requirement.

```
var view = _curry2(function view(lens, x) {
  return lens(Const)(x).value;
});
```

`view`'s second argument, `x`

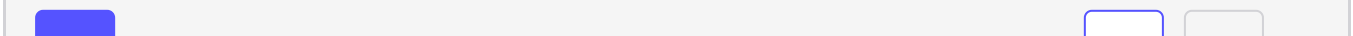
```
function view(lens, x)
```

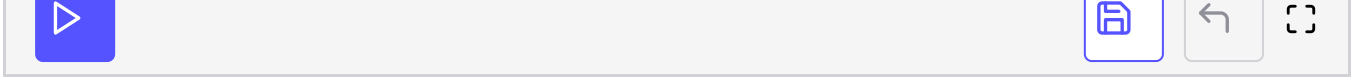
gets forwarded to `lens`

```
lens(Const)(x)
```

Back to the lab!

```
import { assoc, lens, prop, view } from 'ramda';

// Define Const
var Const = function(x) {
  return {
    value: x,
    'fantasy-land/map': function() { return this; }
  };
};

const name = lens(prop('name'), assoc('name'));

// Use it here
const withFunctorFn = name(Const);

// Then pass a target
const withTarget = withFunctorFn({ name: 'Bobo' });

console.log(withTarget);
```



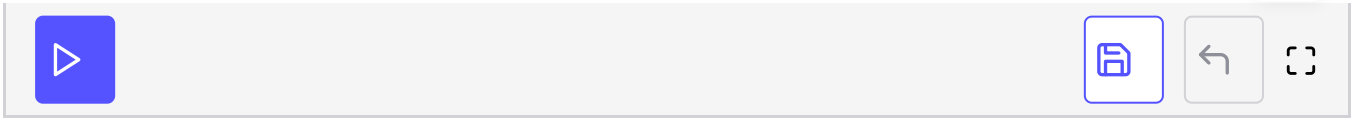


Whoa whoa, look at that!

```
{
  value: 'Bobo',
  'fantasy-land/map': [Function: fantasyLandMap]
}
```

That's our desired value, `Bobo`, after passing through `Const`. It's now a functor!

```
console.log(Const('Bobo'));
```



So we somehow get back a `functor(Bobo)` after this code runs

```
return function(target) {
  return map(
    function(focus) {
      return setter(focus, target);
    },
    toFunctorFn(getter(target))
  );
};
```

After getting its `target`, `lens` begins mapping.

Why map()?

But mapping is to *change* a functor's value. `view`'s only supposed to get a value, not modify it!

`map` does change values, but not if you override it...

Let's dissect the following code snippet.

```
return map(
  function(focus) {
    return setter(focus, target);
  },
  toFunctorFn(getter(target))
);
```

What are the pieces?

```
getter = prop('name');

target = { name: 'Bobo' };

toFunctorFn = function(x) {
  return {
    value: x,
    'fantasy-land/map': function() { return this; }
  };
};
```

Look closer at that functor's `fantasy-land/map` method.

```
`fantasy-land/map`: function() { return this; }
```

Overridden to Do Nothing

It does nothing but return itself.

This makes sense because `view`'s job is to return the data untouched. By returning `this` it effectively ignores `map`.

To contrast, look at `over`'s `toFunctorFn` in the source code.

```
var Identity = function(x) {
  return {value: x, map: function(f) { return Identity(f(x)); }};
};
```

It's an `Identity` functor, similar to the previous lesson!

Instead of doing nothing like `view`, `map`'s being told to transform the lens value according to the supplied function.

```
import { assoc, lens, over, prop, toUpper } from 'ramda';

const name = lens(prop('name'), assoc('name'));
const result = over(name, toUpper, { name: 'Bobo' });

console.log({ result });
```

So whether you're reading with `view` or writing with `set`/`over`, your `toFunctorFn` is the key to making it all work with `map`.

Summary

- Lenses use functors, therefore implement **everything**, even reading a value, with `map`.
- This is possible because an object can define its own `fantasy-land/map` method to take full control of the procedure.
- In `view`'s case, `fantasy-land/map` just returns itself (the requested data).
- In `over`'s case, `fantasy-land/map` transforms the data with a given function.

We're exploring how lenses compose next. That's right, we're going to start combining them to find shocking results!